Introduction

- Operating System: Program that acts as an intermediary between a user and the hardware

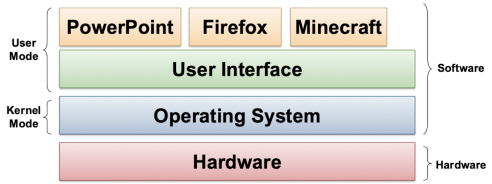
History of OS

- No OS: Programs directly interact with hardware (e.g., plugging wires)
  - Pros: Minimal overhead
  - Cons: Clunky, inefficient
- Batch OS: Executes 1 program (aka job) at a time
  - Used for mainframes: Accepts programs via punch cards
    - Minicomputer: Mini version of mainframe (e.g., Unix)
  - Batch processing: Done in a huge batch (Lots of punchcards -> 1 big tape input -> 1 big tape output)
  - Cons: Inefficient (CPU idle when performing I/O)
- Time-Sharing OS: Allows users to interact with machine using terminals (teletypes)
  - User job scheduling: By switching between apps quickly, creates illusion of concurrency on 1 processor
  - Memory management
  - **Virtualization** of hardware: Each user program (UP) executes as if it has all the resources to itself
- OS on Personal Computer: Machine dedicated to user, not timeshared between multiple users (since it's more affordable)
  - Windows model: 1 user at a time, but can have more than 1 user
  - Unix model: 1 user at machine, but other users can access remotely

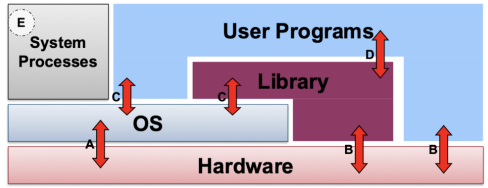
Motivations of OS

- Abstraction: Despite large variation in hardware configurations, OS abstracts away the hardware details and functionalities, so that users can perform essential tasks
- Resource allocator: Since running programs need many resources (e.g., CPU, memory, I/O)
- Control program: Prevents errors and hackers

OS Structure



- OS is a software, just that it runs in kernel mode (i.e., Access to all hardware resources)
- Other software runs in user mode (i.e., Limited access to hardware resources)



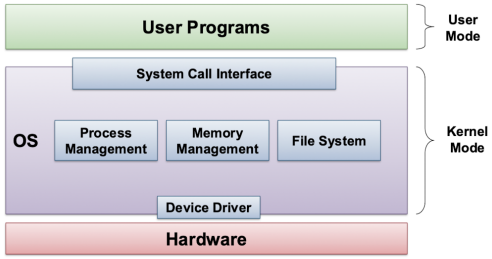
- A, B: Execute machine instructions
- C: Call OS using **system call interface**
- D: UPs call library code (e.g., data structure libraries)
- System processes: Usually part of OS

OS as a Program

- Kernel: Program with some special features:
  - Handles hardware issues
  - Provides system call interface
  - Special code for interruption handlers, device drivers
- Kernel code is different from normal programs, since it has no OS to rely on (i.e., no system calls)
- Many ways to structure an OS

Monolithic OS

> Kernel is 1 big program

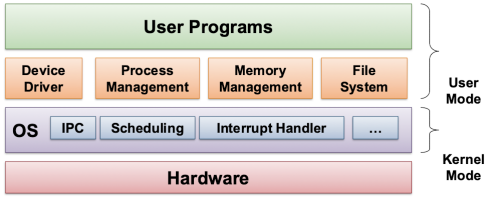


Generic Architecture of Monolithic OS Components

- Good SWE principles still possible with modularization and separation of interfaces
- E.g., DOS, Windows 9x
- Pros: Comprehensive, good performance
- Cons: Highly coupled (1 crash may crash entire kernel), complicated
- Device driver: Developed by hardware provider, so how can we handle incompatibilities?

Microkernel OS

> Kernel is very small and provides only basic functionalities



Generic Architecture of Microkernel OS Components

- Higher-level OS services built **outside** of kernel
- Pros: Robust, extensible, protection between kernel and high-level services (Fewer "blue screen of death")
- Cons: Slow

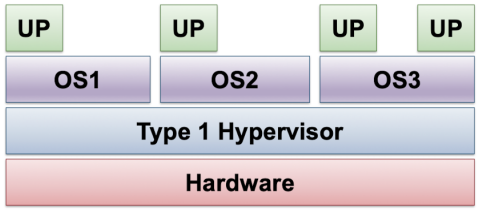
Virtual Machines

> Software emulation of hardware

- Motivation:
  - Since OS assumes total control of hardware, what if we want to run multiple OSes on same hardware at the same time (e.g., cloud computing)
  - OS hard to debug and monitor
- Virtualization of hardware: Similar to virtualization done *by* OS, but now *for* OS
  - OS can run on top of VM
- Created and managed by **Hypervisor** (aka Virtual Machine Monitor (VMM))

Type 1 Hypervisor

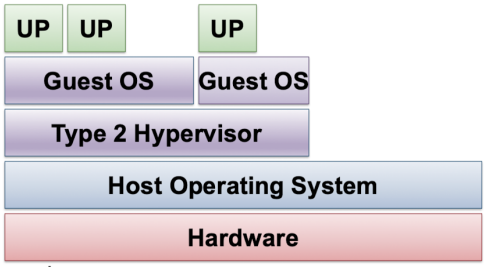
> Runs directly on hardware



- Aka bare-metal hypervisor
- Provides individual VMs to guest OSes via VM interfaces

Type 2 Hypervisor

> Runs in host OS



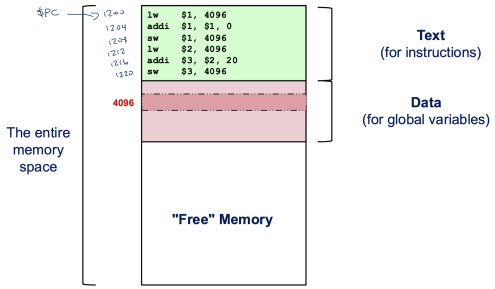
- Guest OS runs inside VM

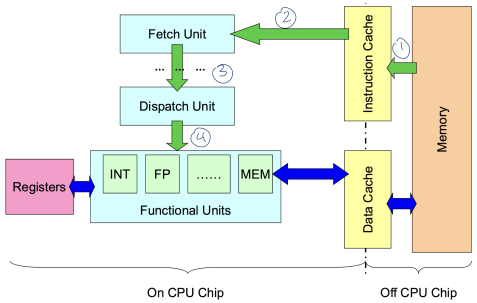
Process Abstraction

- To execute a program, need following information:
  - Memory context: Text, Data, Stack, Heap
  - Hardware context: General purpose registers, Program Counter (PC), Stack Pointer, Frame Pointer
  - OS context: Process ID, Process State

Recap: Hardware and Memory Context

- To execute a program, C code -> Assembly code
  - **Text** for instructions and **data** for global variables stored in memory





- Memory: Storage for instruction and data
- Cache: Duplicate part of memory for faster access
- Fetch Unit: Loads instruction from memory
  - Program Counter (PC): Special register indicating address of instruction to run
- Functional Units: Execute instructions
- Registers: Internal storage for fastest access speed
  - **General Purpose Register** (GPR): Accessible by user program
  - Special Register (e.g., Program Counter)

Stack Memory

> Handling function calls

Motivation

- Before: 1 huge chunk of C code

```c
int i = 0;
i = i + 20;
```

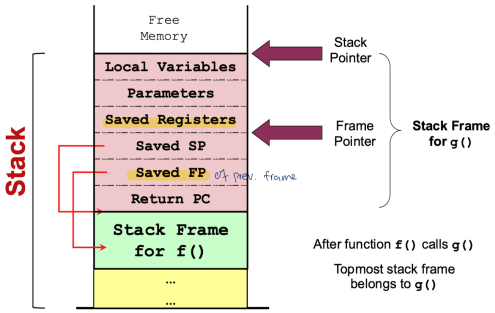
- Now: What if we use functions?

```c
int g(int i, int j) {
    int a = i + j;
    return a;
}
```

- Problems:
  - Control flow: How to jump in when calling function and jump out after function call is done?
  - Data storage:
    - How to allocate memory space for **local variables** `i`, `j`, and `a`?
    - How to store return result?
- Can we just use data memory space? No, cannot differentiate between multiple calls of the same function

Solution

- Stack Memory Region: New memory region to store information for function invocations
  - `main` method included inside
- Stack Frame: Information of 1 function invocation
  - Frame added on top when function is called; Frame removed from top when function call ends
  - Stores:
    - Local variables
    - Parameters
    - Saved registers
    - Saved Stack Pointer
    - Saved Frame Pointer
    - Return Program Counter

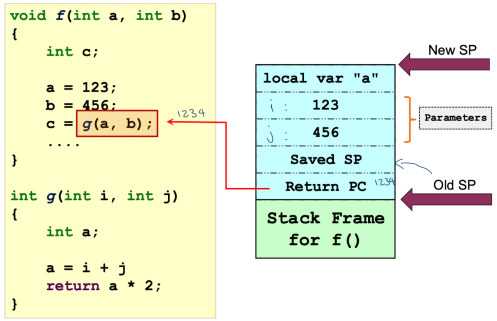


- Stack Pointer: Top of the stack region (1st unused location) that **dynamically changes** as we add more local variables and parameters
  - Special register

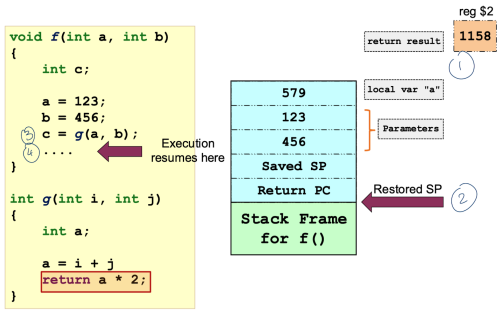
- Requires stack frame to store previous frame's SP
- Function Call Convention: Different ways to setup stack frame
  - E.g., is the information stored in stack frame or register? Prepared by caller or callee?
  - No correct answer, as long as consistent

Sample Frame Setup and Teardown

- On function call:
  1. Caller: Pass arguments with registers and/or stack
  2. Caller: Save Return PC on stack
  3. Transfer control from caller to callee
  4. Callee: Save registers used by callee
  5. Callee: Save old Frame Pointer and Stack Pointer
  6. Callee: Allocate space for local variables on stack
  7. Callee: Adjust Stack Pointer and Frame Pointer **in register**



- On function return:
  1. Callee: Place return result in register (if applicable)
  2. Callee: Restore saved registers, FP, SP
  3. Transfer control from callee to caller using saved PC
  4. Caller: Use return result (if applicable)
  5. Caller: Continues execution in caller



- Is stack frame deleted explicitly? No, stack pointer suffices
  - Also why local variable needs to be initialized with new value. Without initialization, repeated calls to same function may mistakenly set variable to value from previous call

Frame Pointer

> To facilitate access of stack frame items

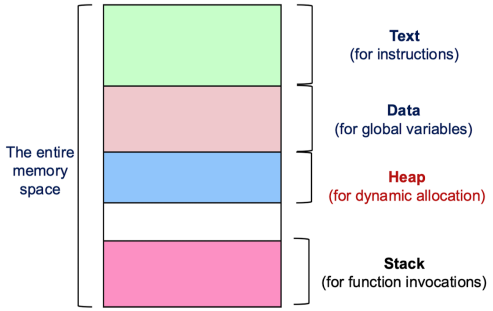
- Problem: Stack Pointer is hard to use, since it's dynamically changing from adding local variables -> Need to dynamically change offset
- Solution: Frame Pointer
  - Frame Pointer: Points to fixed location in stack frame
  - Other items have fixed displacement from FP
  - Requires stack frame to store previous frame's FP
  - Platform dependent
  - FP only for compiler's convenience

Saved Registers

- Problem: Number of General Purpose Registers limited
- Solution: Register spilling
  - When GPRs are all used up, use memory to temporarily hold GPR values
  - GPR can then be reused for other purposes
  - GPR values can be restored later
- Saved registers: Saved values from registers that the function intends to use before function starts
  - On function return, restore those registers
  - Stored in stack frame

Heap Memory

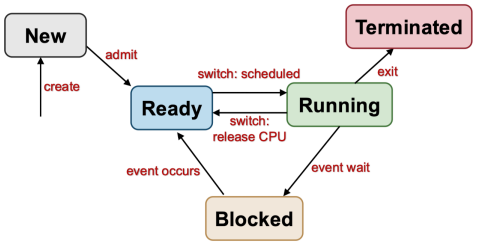
> Dynamically allocated memory (i.e., needed during execution time)



- Examples:
  - In Java, `new`
  - In C, `malloc()`
- Why not use data memory? Allocated only at runtime, so size not known during compilation
  - E.g. size of array not known until runtime
- Why not use stack memory? No definite deallocation time
- Problems:
  - Trickier to manage due to variable size
  - "Holes" in the memory

Process Management

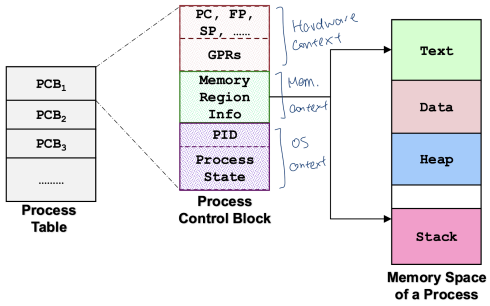
- Motivation: Need to allow many programs to share the hardware for efficient utilization
- Solution: OS needs to allow switching from program A to B, which needs to:
  - Store information about program A
  - Replace program A's information with program B
- Process: Dynamic abstraction to describe a running program
  - Aka task or job
  - Includes memory context, hardware context, and OS context
  - Process ID (PID): Number that distinguishes processes from each other
    - OS dependent issues: Are PIDs reused? Maximum number of processes? Reserved PIDs?
  - Process State: Indication of execution status of process
    - Process Model: Set of **states** and **transitions** that describes behaviors of a process



- Given n processes:
  - With 1 CPU core:
    - ≤ 1 process in running state
    - 1 transition at a time
  - With m CPUs:
    - $\leq m$ processes in running state
    - Possibly parallel transitions
- Assumption in CS2106: Only 1 core!

Process Table

> Putting it all together

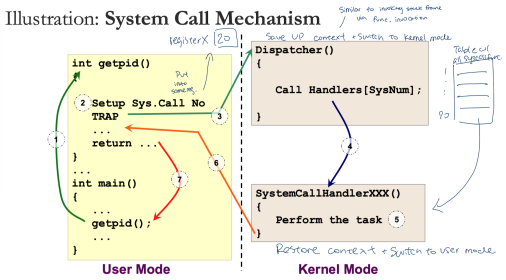


- Process Control Block (PCB): Data structure for storing entire execution context of a process
  - Aka Process Table Entry
  - Kernel maintains PCB for all processes as 1 table (i.e., process table)
- Problems:
  - Scalability: How many concurrent process can you have?
  - Efficiency: How to ensure minimum space wastage?

System Calls

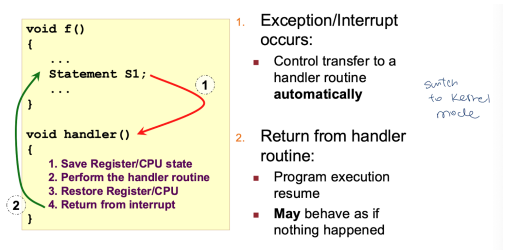
> API for OS

- Provides way of calling services in kernel
  - Different from normal function call: Need to change from user mode to kernel mode
- APIs are OS-specific
  - E.g., Unix system calls in C/C++
    - Function wrapper: Library functions that have same name and parameters as system calls
      - E.g., `getpid()`
    - Function adapter: Library functions that present a more user-friendly version of system call
      - E.g., `printf()` library call uses `write()` from system call



1. UP calls library call
2. Library call places **system call number** in designated location (e.g., register)
3. Library call switches from user mode to kernel mode using special instruction (Aka TRAP) and saves CPU state
4. In kernel mode, **dispatcher** determines system call handler using system call number as index
5. System call handler executed
6. System call handler restores CPU state, returns to library call, and switches from kernel mode back to user mode
7. Library call returns to user program

Exception and Interrupt



- Exception: Can happen when executing machine level instructions
  - E.g., Arithmetic Errors
  - Synchronous: Due to program execution
  - Effect: Have to execute **exception handler** in kernel automatically
- Interrupt: Can happen due to external events
  - Usually hardware related (e.g., keyboard)
  - Asynchronous: Independent of program execution
  - Effect: Have to execute **interrupt handler**

Process Abstraction in Unix

```
ps // Process status
ps -e // List all processes
man fork // Help manual
```

Creating a New Process

```
#include <sys/types.h>
#include <unistd.h>

int fork(); // syntax
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    printf("I am ONE\n");
    fork();
    printf("I am seeing DOUBLE\n"); // Printed twice
    return 0;
}
```

- Behavior: Parent process -> Parent and child processes
  - Child process is **duplicate copy** of current executable image (i.e., same contexts)
  - Child only differs in: PID, Parent PID (PPID), `fork()` return value
  - Order of child and parent is non-deterministic
- Returns:
  - For parent process: PID of the child
  - For child process: 0
  - Use return value to distinguish parent and child
- The Master Process: The ancestor of all processes (`init` process)
  - Created in kernel at boot up time
  - Traditionally, PID = 1

Passing Arguments to a Program

```
int main(int argc, char* arv[]) {}
```

- `argc`: Number of command line arguments, including program name itself
  - E.g., `a.exe 1 2 3 hello` -> 5 arguments

Executing Another Program

- Motivation: `fork()` itself is not useful, since still need to provide full code for child process

```
#include <unistd.h>

// syntax
int execl(const char *path,
          const char *arg0,
          ...,
          const char *argN, NULL);
```

- `path`: Location of executable
- `arg0, ..., argN`: Command line arguments
- `NULL`: To indicate end of argument list

```
#include <unistd.h>

// Same as running: ls -al
int main() {
    printf(...);
    execl("/bin/ls", "ls", "-al", NULL);
    printf(...); // Will not run! Since replaced
}
```

- Behavior: Replaces current executing process image (i.e., code) with new one
  - Process-related information still same
- Combining `fork()` and `execl()`: Can spawn child process and let it perform a task, while parent process is still around to accept another request

Process Termination

```
#include <stdlib.h>

void exit(int status); // syntax
```

- `status`: Returned to parent process
  - 0: Successful execution -> Normal termination
  - Else: Indicate problematic execution
- Behavior:
  - Most system resources used by process released by `exit()`
  - Some basic process resources not releasable -> Zombie State
    - Motivation: What if parent asks for child PID and status code in `wait()` call?
- Zombie Process: Child process terminates before parent, but parent has not called wait
  - Cannot be killed
- Orphan Process: Parent process terminates before child process
  - `init` process becomes pseudo parent
  - Child termination sends signal to `init`, which uses `wait()` to clean up
- Implicit `exit()`: Return from `main()` implicitly calls `exit()`
  - If `main()` returns some number, then that number is the status code

Parent-Child Synchronization

> Parent process can wait for child process to terminate

```
#include <sys/types.h>
#include <sys/wait.h>

int wait(int *status); // syntax
```

- `status`: Pointer that stores and points to the exit status of terminated child process
  - Pass in `NULL` if don't want this
  - `$?`: Application of this in shell -> Holds exit status of last executed command
- Returns: PID of terminated child process

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    if (fork() == 0)
        printf("Hello from child\n");
    else {
        printf("Hello from parent\n");
        wait(NULL);
        printf("Child has terminated\n");
    }
    printf("Bye\n");
    return 0;
}

// Case 1: If child runs first
```

```
// Hello from child
// Bye
// Hello from parent
// Child has terminated
// Bye

// Case 2: If parent runs first
// Hello from parent
// Hello from child
// Bye
// Child has terminated
// Bye
```

- Behavior:
  - Blocks parent process until at least 1 child terminates
  - Cleans up child system resources not removed on `exit()` (including zombie processes)
- Variations:
  - `waitpid()`: Wait for specific child process
  - `waitid()`: Wait for any child process to change status
- If `wait()` is run in child process, basically ignored, since no child processes to wait for

Implementing `fork()`

- Simple way: Make almost exact copy of parent PCB
  - Problem: Memory copy very expensive
  - Observation: If only read, can share with parent; if need write, then a copy is needed
- Copy on Write: Only duplicate memory location when it is written to; otherwise, parent and child share the same memory location
  - More optimized
- `clone()` provides more control on what to copy than `fork()`, which copies everything

Process Scheduling

> Given many ready processes, which should be chosen to run?

- Concurrent processes: Multiple processes progress in execution at the same time
  - Virtual parallelism: 1 core
    - Timeslicing: Interleave instructions from both processes
      - OS incurs overhead in context switching when switching processes
  - Physical parallelism: Multi-core
- Scheduler: Part of the OS that makes scheduling decision
  - Scheduling algorithm: Algorithm used by scheduler
- Process behavior: Combination of CPU-activity and IO-activity
- Processing environment: Influences choice of scheduling algorithm
  - Batch processing: No user interaction required
  - Interactive: Active user interacting with system; should be responsive
  - Real-time processing: Have deadline to meet; periodic
- Common evaluation criteria:
  - Fairness: Each process gets fair share of CPU time
    - No starvation (Process ready, but cannot ever run)
  - Utilization: All parts of computing system should be used
- When to perform scheduling?
  - Non-preemptive (Cooperative): Process stays running until it blocks or gives up (**yields**) CPU voluntarily
  - Preemptive: Process given **fixed time quota** to run

Scheduling for Batch Processing

- Criteria for batch processing:
  - Turnaround time: Total time taken for process to finish
    - Waiting time: Time spent waiting for CPU
  - Throughput: Number of tasks finished per unit time
  - CPU utilization: Percentage of time when CPU is working on a task

First-Come First Served (FCFS)

- Idea:
  - Ready tasks stored in queue based on arrival time
  - Pick first task in queue to run until task is done or blocked
  - Blocked task removed from queue
- Pros: No starvation
- Cons:
  - Average waiting time can be reduced (SJF)
  - Convoy Effect: Cannot efficiently distribute CPU and IO tasks -> CPU/IO usually idling
    - E.g. 1 CPU task, followed by many IO tasks

Shortest Job First (SJF)

- Idea: Pick task with smallest total CPU time
- Pros: Given fixed set of tasks, minimizes average waiting time
- Cons:
  - Need to know task's total CPU time in advance
  - Starvation possible: Since short jobs always prioritized first and long jobs can starve
- Predicting CPU time using exponential average: $\text{Predicted}_{n+1} = \alpha \text{Actual}_n + (1-\alpha)\text{Predicted}_n$
  - Task usually has many phases of CPU-activity (Interleaved with IO-activity) $\rightarrow$ Use previous predicted and actual CPU-activity time to predict future

Shortest Remaining Time (SRT)

- Idea: Pick task with shortest remaining time
  - Variation of SJF
  - Preemptive: New jobs with shorter remaining time can cut off running job
- Pros and cons: Same as SJF

Scheduling for Interactive Systems

- Criteria for interactive systems:
  - Response time: Time between arrival and first response by system (Since interactive)
  - Predictability: Variation in response time; Lesser variation $\rightarrow$ More predictable
- Idea: To ensure good response time, **scheduler needs to run periodically to preempt**
  - Timer interrupt: Interrupt that goes off periodically based on hardware clock
  - Interval of Timer Interrupt (ITI): Interval for when OS scheduler is invoked
  - Time Quantum: Execution duration given to process
    - Must be multiples of ITI

Round Robin (RR)

> FCFS with time quantum

- Idea:
  - Ready tasks stored in queue based on arrival time
  - Pick first task in queue to run until task is done or blocked or **time quantum elapsed**
  - Blocked task removed from queue
- Choice of time quantum:
  - Big quantum: Better CPU utilization, but longer waiting time
  - Small quantum: More overhead and worse CPU utilization from context switching, but shorter waiting time
- Response time guaranteed: Since given n tasks and quantum q, response time bounded by $(n-1)q$
- Timer interrupt needed

Priority-Based Scheduling

- Idea:
  - Assign priority value to all tasks
  - Pick task with highest priority value
- Variants:
  - Preemptive: Higher priority processes can preempt running process with lower priority
  - Non-preemptive: Higher priority processes need to wait for next round of scheduling
- Cons:
  - Low priority processes can starve
    - Solution: Can decrease priority of running process after every time quantum
    - Solution: Can remove current running process in next round of scheduling after quantum
  - Hard to control CPU time for a process
- Priority Inversion: Higher priority task held up by lower priority task
  - E.g. Priority: $A = 1, B = 3, C = 5$
    - Task C starts and locks resource
    - Task B arrives and preempts C due to priority
    - Task A arrives and needs same resource as C
    - Task A blocked by C despite higher priority

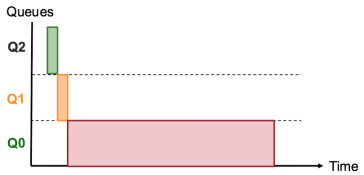
Multi-Level Feedback Queue (MLFQ)

> Lower priority for long-running jobs

- Motivation: How to schedule without perfect knowledge (e.g., process behavior, running time)?
- Adaptive: Learns the process behavior during execution
- Rules:
  - Priority of A $>$ Priority of B $\rightarrow$ A runs
  - Priority of A $==$ Priority of B $\rightarrow$ A and B runs in Round Robin
  - New job given highest priority
  - Job fully utilizes time quantum $\rightarrow$ Priority reduced
  - Job gives up or blocked before time quantum $\rightarrow$ Priority stays the same

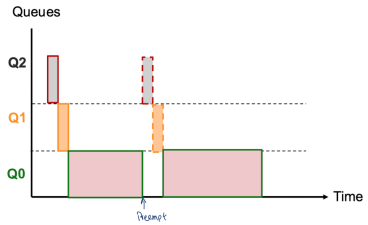
MLFQ: Example 1

- 3 Queues: Q2 (highest priority), Q1, Q0
- A single long running job
  - Try to apply the rules and check your understanding



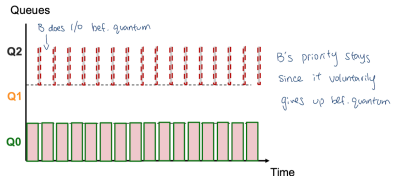
MLFQ: Example 2

- Example 1 + a short job in the middle
  - A short job appears sometime in the middle



MLFQ: Example 3

- Two jobs:
  - A = CPU bound (already in the system for quite some time)
  - B = I/O bound



- Pros:
  - Minimizes response time for IO bound processes
  - Minimizes turnaround time for CPU bound processes
- Cons:
  - Starvation for long-running jobs: Too many interactive jobs
  - Program can change behavior (e.g., CPU bound for a long time $\rightarrow$ Interactive): Low response time
    - Solution: Can periodically bump all jobs to highest priority
  - Abuse I/O: Do an I/O right before quantum ends, so priority remains same forever and hog the CPU
    - Solution: Scheduler can track total time a task has used at a priority level and reduce priority once task hits the limit

Lottery Scheduling

- Idea:
  - Give out "lottery tickets" to processes for various resources
  - When scheduling decision needed, a ticket is chosen randomly and winner is granted the resource
  - In the long run, a process holding $X\%$ of tickets $\rightarrow$ Wins $X\%$ of scheduling $\rightarrow$ Uses resource $X\%$ of the time
- Pros:
  - Responsive: New process can join next lottery
  - Can prioritize: Important process can be given more tickets
  - Process can distribute tickets to child processes
  - Each resource has own set of tickets

Inter-Process Communication

- Motivation: Hard for cooperating processes to share information, since memory space is independent
  - So far, only have `wait()`: But that's only at the end of the process and it can only pass a status code

Shared Memory

- Idea:
  1. Process P_1 creates **shared memory region** M
  2. Process P_1 and P_2 attach memory region M to its own memory space
  3. P_1 and P_2 communicate via M
- Master program: Creates and tears down shared memory region

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

int main() {
    int shmid, i, *shm;

    // Create shared memory region
    shmid = shmget(IPC_PRIVATE, 40, IPC_CREAT | 0600);
    if (shmid == -1) {
        printf("Cannot create shared memory!\n");
        exit(1);
    } else {
        printf("Shared memory id = %d\n", shmid);
    }

    // Attach master program to shared memory region
    shm = (int*) shmat(shmid, NULL, 0);
    if (shm == (int*) -1) {
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
```

```
    // First element in shared memory region used as control value
    // 0: Not ready, 1: Values ready
    shm[0] = 0;
    while (shm[0] == 0) {
        sleep(3);
    }
    for (i = 0; i < 3; i++) {
        printf("Read %d from shared memory\n", shm[i+1]);
    }

    // Detach and destroy shared memory region
    shmdt((char*) shm);
    shmctl(shmid, IPC_RMID, 0);

    return 0;
}
```

- Slave program: Attaches and writes values to shared memory region

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

int main() {
    int shmid, i, input, *shm;

    // Receive shared memory id as input
    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

    // Attach
    shm = (int*) shmat(shmid, NULL, 0);
    if (shm == (int*) -1) {
        printf("Cannot attach!\n");
        exit(1);
    }

    // Write values into shared memory
    for (i = 0; i < 3; i++) {
        scanf("%d", &input);
        shm[i+1] = input;
    }

    // Let master program know we're done
    shm[0] = 1;

    // Detach
    shmdt((char*) shm);

    return 0;
}
```

- Pros:
  - Efficient: Only creating and attaching shared memory region involves OS
  - Ease of use: Shared memory space behaves the same as normal memory space
- Cons:
  - Lack of synchronization when accessing shared resources
    - E.g. How to ensure master program sets control value to 0 before slave program?
  - Hard to implement

Message Passing

- Idea:
  1. Process P_1 prepares message M and sends it to process P_2
  2. Process P_2 receives message M
- Send (`send()`) and receive (`recv()`) operations go through OS
  - M in P_1's PCB $\rightarrow$ M in kernel memory $\rightarrow$ M in P_2's PCB
- Naming Scheme:
  - Direct communication: Sender and receiver explicitly name other party
    - Need to know identity of other party
    - Unix: Unix domain socket
  - Indirect communication: Messages sent to and received from shared mailbox or port
    - 1 mailbox can be shared among many processes
    - Unix: Message queue
- Synchronization:
  - Blocking primitive: Receiver blocked until message arrived (Synchronous)
  - Non-blocking primitive: Receiver either receives message if available or some indication that message is not ready yet (Asynchronous)
- Pros:
  - Portable: Can work on different processing environments
  - Easier synchronization: Through blocking primitive
- Cons:
  - Inefficient: Need OS intervention and extra copying in kernel memory

Unix Pipe

> Link input and output channels of processes

- Process has 3 default communication channels:
  - Standard in: `stdin`; In C, `scanf()`
  - Standard out: `stdout`; In C, `printf()`
  - Standard error: `stderr`; In C, `perror()`
- Unix shell: `|`
- Producer-Consumer relationship:

    - P produces/writes n bytes
    - Q consumes/reads m bytes
- Queue behavior: Must access data in order

Creating a Pipe

```
#include <unistd.h>

int pipe(int pipefd[2]); // syntax
```

- Parameter: Array of 2 file descriptors
    - `pipefd[0]`: Read end of pipe
    - `pipefd[1]`: Write end of pipe
- Returns: 0 for success and !0 for errors

```
#define READ_END 0
#define WRITE_END 1

int main() {
    int pipeFd[2], pid, len;
    char buf[100], *str = "hello!";

    pipe(pipeFd);
    if ((pid = fork()) > 0) {
        // Parent process
        close(pipeFd[READ_END]);
        write(pipeFd[WRITE_END], str, strlen(str) + 1);
        close(pipeFd[WRITE_END]);
    } else {
        // Child process
        close(pipeFd[WRITE_END]);
        len = read(pipeFd[READ_END], buf, sizeof(buf));

        printf("Process %d read: %s\n", pid, buf);
        close(pipeFd[READ_END]);
    }
}
```

- `read()`: Blocking call until parent writes
- `close()`: Closes reading/writing end for process
    - Signal end of reading/writing

Unix Signal

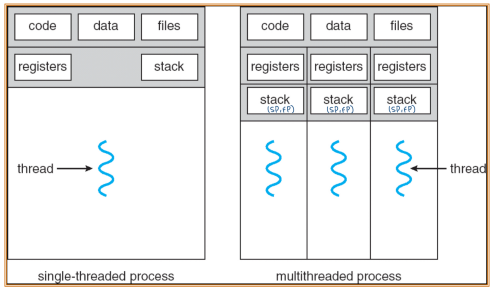
> Asynchronous notification about an event that is sent to a process/thread

- Recipient must handle signal by using default handler or providing custom handler
- Common signals in Unix: Kill, interrupt

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void customHandler(int signo) {
    if (signo == SIGSEGV) {
        printf("Memory access blew up!\n");
        exit(1);
    }
}

int main() {
    int *ip = NULL;

    // Register our own handler to replace default handler
    if (signal(SIGSEGV, customHandler) == SIG_ERR) {
        printf("Failed to register handler\n");
    }

    // Will cause segmentation fault
    *ip = 123;

    return 0;
}
```

Threads

- Motivation:
    - Process creation is expensive
        - Duplicate memory space and process context
        - Context switching
    - Hard for independent processes to do IPC
- Idea:
    - Traditional process has **single thread of control**: Only 1 instruction in program is running $\rightarrow$ 1 PC
        - Multi-process model: Multiple "threads of control" using `fork()`
    - Add more threads of control to same process $\rightarrow$ Many PCs



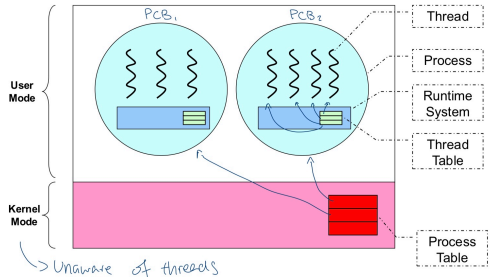
> Thread is a "lightweight process"

- Process vs. Thread:
  - Multithreaded process: Single process with multiple threads
  - Threads in the same process share:
    - Memory context: Text, data, heap
    - OS context: PID, files
  - Unique information specific to each thread:
    - Thread ID
    - General purpose registers: For calculations
    - Special registers: PC
    - "Stack": Stack still shared within process, just FP and SP copied
- Pros:
  - Less resources needed
  - No need for IPC
  - More responsive, since no need context switching
  - Scalable, since multithreaded programs can take advantage of multiple CPUs
- Cons:
  - Parallel execution of multiple threads → Parallel system call possible → Need to guarantee correct behavior
  - Impact on whole process
    - `fork()` duplicates process, then threads inside?
    - Single thread calls `exit()`, then the other threads inside the process?

Thread Models

User Thread

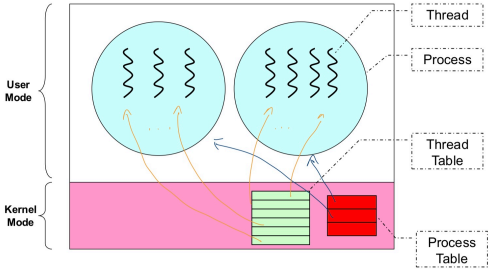
> Implemented as a user library, so kernel is not aware of threads



- Pros:
  - Can have multithreaded programs on any OS
  - Thread operations just library calls
  - More configurable and flexible
- Cons:
  - Scheduling done at process level
    - 1 thread blocked → Entire process blocked
    - Cannot exploit multiple CPUs, since scheduling done by process

Kernel Thread

> Implemented in the OS



- Pros:
  - Kernel can schedule by threads, instead of by process
    - Threads in same process can run simultaneously on multiple CPUs
- Cons:
  - Thread operations now a system call → Slower and more resource intensive
  - Less flexible, since implemented in kernel and used by all multithreaded programs
    - If too many features, then expensive and overkill for simple program
    - If too little features, then not flexible for some programs

Hybrid Thread

- Have both kernel and user threads
- User threads can bind to kernel thread
- OS schedules on kernel threads only
- Pros: More flexible, since can limit concurrency on any process

POSIX Threads

- Defines API, not implementation
  - Can be implemented as user or kernel thread

Creating and Terminating Threads

```
#include <pthread.h>

gcc XXX.c -lpthread

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

- Returns: 0 if success; !0 if errors
- Parameters:
  - `tidCreated`: Thread ID for created thread
  - `threadAttributes`: Control the behavior of new thread
  - `startRoutine`: Function pointer to the function to be executed by thread
  - `argForStartRoutine`: Arguments for the `startRoutine` function
- Different from `fork()`, since forked child starts at same location as parent

```
// syntax
int pthread_exit(void* exitValue);
```

- `exitValue`: Value to be returned to whoever syncs with this thread
- If `pthread_exit()` is not used, thread will end automatically at the end of `startRoutine`
  - If `return XYZ;` is used, then `XYZ` is captured as `exitValue`

```
int globalVar;

// void*: Generic pointer
void* doSum(void* arg) {
    int i;
    for (i = 0; i < 1000; i++)
        globalVar++;
}

int main() {
    pthread_t tid[5];
    int i;
    for (i = 0; i < 5; i++)
        pthread_create(&tid[i], NULL, doSum, NULL);
    return 0;
}
```

- Final sum may not be 5000!
  - Possible for `main()` to return first, before all the calculations

Simple Synchronization

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

- Behavior: Waits for termination of another `pthread`
- Returns: 0 if success; !0 if errors
- Parameters:
  - `threadID`: Thread ID of `pthread` to wait for
  - `status`: Exit value returned by target `pthread`

```
int globalVar;

void* doSum(void* arg) {
    int i;
    for (i = 0; i < 1000; i++)
        globalVar++;
}

int main() {
    pthread_t tid[5];
    int i;
    for (i = 0; i < 5; i++)
        pthread_create(&tid[i], NULL, doSum, NULL);

    for (i = 0; i < 5; i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```

- Slightly better, but final sum still may not be 5000!
  - `globalVar++` is not atomic, since it involves read → calculate → write

Synchronization

Problems with Concurrent Execution

- Race condition: Execution outcome depends on order in which shared resource is accessed/modified
  - Concurrent execution may be non-deterministic (Like the above example)
- Incorrect execution due to unsynchronized access to shared modifiable resources
- Solution:
  - Critical section: Code segment with race condition
  - At any time, only 1 process can execute in critical section and other processes are prevented from entering the same critical section

Critical Section

- Properties of correct implementation:
  - Mutual exclusion: If there is a process executing in critical section, then all other processes are prevented from entering
  - Progress: If no process is in critical section, then a waiting process should be granted access
  - Bounded wait: After process requests to enter critical section, there exists a limit on number of times other processes can enter critical section beforehand
    - I should eventually enter!
  - Independence: Process not in critical section should not block other processes
- Symptoms of incorrect synchronization:
  - Deadlock: All processes blocked
  - Livelock: Processes keep changing state to avoid deadlock and make no progress
    - E.g. 2 people in sidewalk and both move in same direction
  - Starvation: Some process forever blocked

Assembly Level Implementation

```
TestAndSet Register, MemoryLocation
```

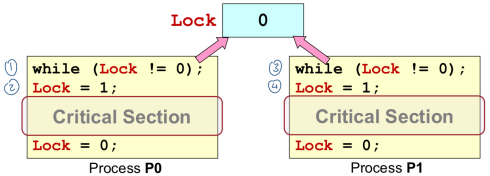
- Behavior: **Atomic**
  1. Load content at `MemoryLocation` into `Register`
  2. Stores 1 into `MemoryLocation`

```
void EnterCS(int* Lock) {
    while (TestAndSet(Lock) == 1);
}

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

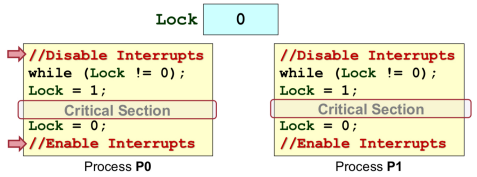
- Assume `TestAndSet` has high-level language version:
  - Parameter: Memory address M
  - Saves content at M to register and sets content of M to 1
  - Returns register value
- Intuition: 1st process can escape `while` loop while other processes are stuck in loop until lock is set to 0 again
- Cons: Busy waiting (Keeps checking condition) → Wasteful processing power

High-Level Language Implementation

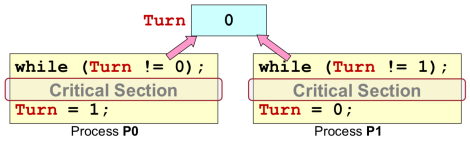
- Attempt 1: Does not work



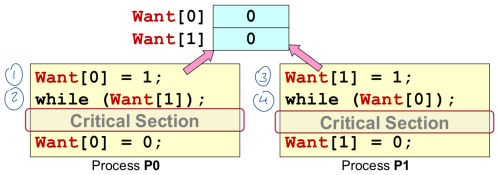
- If in sequence 1 → 3 → 2 → 4, violates mutual exclusion
  - Because load and store not atomic, unlike `TestAndSet`
- Attempt 1a: Works, but...



- Intuition: Disable interleaving
- Cons:
  - Buggy critical section may stall whole system, since cannot preempt
  - Busy waiting
  - Need permission to disable and enable interrupts
- Attempt 2: Does not work

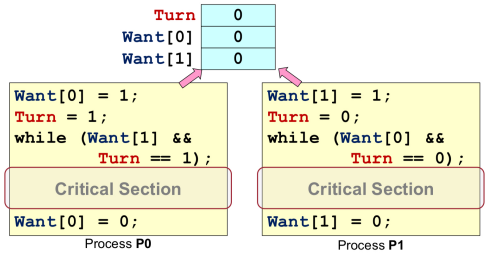


- Starvation: If P0 never enters critical section, then P1 starves
  - Violate independence
- Attempt 3: Does not work



- If in sequence $1 \to 3 \to 2 \to 4$, deadlock
  - Progress violated

Peterson's Algorithm



- Assumption: Write to `Turn` is atomic
- Intuition: Attempt 3 with deadlock solved
  - Solves deadlock with `Turn`, since `Turn` is either 0 or 1 when we reach `while`
  - Allows other process to enter once done, since `want[1] = 0` now
- Cons:
  - Busy waiting
  - Low level
  - Not general: Only for synchronization of 2 processes

Semaphore

> Data structure with 2 atomic operations: `wait()` and `signal()`

- Generalized synchronization mechanism
  - Only behavior specified; can have different implementations
- Semaphore S contains:
  - Capacity: Integer value that is initialized to any non-negative values
  - List to track waiting processes
  - **Atomic** operations:
    - `wait(S)`: Aka `P()` or `Down()`
      1. If $S \leq 0$, then blocks process/go to sleep
      2. Decrement S
    - `signal(S)`: Aka `V()` or `Up()`
      1. Increment S
      2. Wakes up a sleeping process to try again
- Given $S_{\text{initial}} \geq 0$, $S_{\text{current}} = S_{\text{initial}} + \texttt{\#signal(S)} - \texttt{\#wait(S)}$
  - `#signal(S)`: Number of `signal()` operations executed
  - `#wait(S)`: Number of `wait()` operations **completed**
- General semaphore: $S \geq 0$
- Binary semaphore: $S = 0$ or $S = 1$
  - Can build general semaphores
- Mutex: Usage of synchronization mechanism (e.g., semaphore) to ensure mutual exclusion

```
Wait(S);
// Critical section
Signal(S);
```

- Semaphores ensure mutual exclusion
  - N_{CS} = Number of processes in CS = Number of processes that completed `wait()` but not `signal()` = $\texttt{\#wait(S)} - \texttt{\#signal(S)}$
  - $S_{\text{current}} + N_{CS} = S_{\text{initial}} = 1$
  - $N_{CS} \leq 1$ since $S_{\text{current}} \geq 0$
- A semaphore prevents deadlock
  - Deadlock means all processes stuck at `wait()` $\to S_{\text{current}} = 0$ and $N_{CS} = 0$
  - But $S_{\text{current}} + N_{CS} = 1$
- But deadlock still possible with multiple semaphores
  - E.g. need to acquire resources from 2 semaphores P and $Q \to$ `wait()` in different order
  - Solution: Ensure order is same
- A semaphore prevents starvation
  - Suppose $P1$ blocked at `wait()`
  - $P2$ is in CS, and exits CS with `signal()`
    - If no other processes sleeping, $P1$ wakes up
    - If there are other processes, $P1$ eventually wakes up

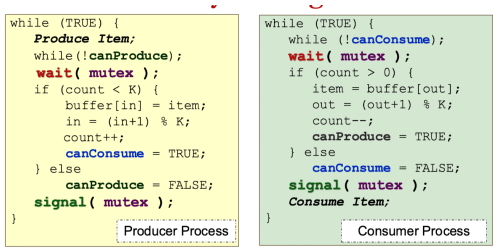
Classic Synchronization Problems

Producer Consumer

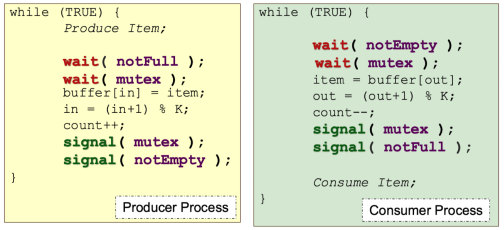
- Processes share a bounded buffer of size K
- **Producer** processes insert items in buffer
  - Only when buffer is not full
- **Consumer** processes remove items from buffer
  - Only when buffer is not empty

```
// Producer process
while (True) {
    // Produce item
    if (count < K) {
        buffer[in] = item;
        in = (in + 1) % K;
        count++;
    }
}

// Consumer process
while (True) {
    if (count > 0) {
        item = buffer[out];
        out = (out + 1) % K;
        count--;
    }
    // Consume item
}
```

- Problems:
  - No synchronization: When running concurrently, there exists critical sections
  - Produced item wasted if buffer full
  - Consume nothing if buffer empty
- Solution 1: Busy waiting



- Intuition:
  - Wraps critical section in mutex
  - `canProduce`: Flag for blocking through busy waiting to not waste produced item
  - `canConsume`: Flag for blocking to not consume nothing
- Solution 2: No busy waiting with more semaphores!



- Intuition:
  - `notFull` semaphore: Forces producer to sleep when buffer full
    - Initialized to K
  - `notEmpty` semaphore: Forces consumer to sleep when buffer empty
    - Initialized to 0

Readers Writers

- Processes share data structure D
- Reader processes retrieve information from D
  - Can share access with other readers
- Writer processes modify information in D
  - Must have exclusive access
- Attempt 1: Does not work

```
// Writer process
while (True) {
    wait(roomEmpty);
    // Modify data
    signal(roomEmpty);
}

// Reader process
while (True) {
    nReader++;
    if (nReader == 1)
        wait(roomEmpty);

    // Read data

    nReader--;
    if (nReader == 0)
        signal(roomEmpty);
}
```

- Intuition:

- `roomEmpty` semaphore: Blocks writer if room is not empty and blocks first reader if room is not empty
  - Initialized to 1
- Problems:
  - Critical sections of `nReader++` and `nReader--`
  - If first reader is waiting on writer, and second reader comes in, it will skip `wait(roomEmpty)`
- Solution: Add a mutex

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

```
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

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

- Intuition: Adding mutex solves both problems, since first reader waiting for writer to finish will block other readers too
- Writer Starvation: In the above solution, as long as at least one reader, writers cannot write and starve
  - Unbounded wait time
  - Solutions:
    - If writer is waiting, new readers are blocked until writer finishes
    - Implement timeout for readers

Dining Philosophers

- 5 philosophers seated around a table and 5 **single** chopsticks placed between each pair of philosophers
- Philosopher is either thinking, hungry, or eating
- To eat, a philosopher must acquire both chopsticks to his/her left and right
- Attempt 1: Does not work

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

// For philosopher i
while (True) {
    // Think
    takeChopstick(LEFT);
    takeChopstick(RIGHT);
    // Eat
    putChopstick(LEFT);
    putChopstick(RIGHT);
}
```

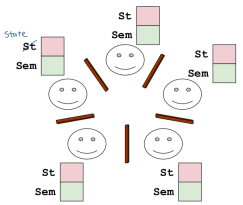
- Problem: Deadlock, when all philosophers simultaneously takes up left chopstick, and none can continue
  - To fix this, make philosopher put down left chopstick if right chopstick cannot be acquired → No deadlock, but possible livelock
- Attempt 2: Add a mutex

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

// For philosopher i
while (True) {
    // Think
    wait(mutex);
    takeChopstick(LEFT);
    takeChopstick(RIGHT);
    // Eat
    putChopstick(LEFT);
    putChopstick(RIGHT);
    signal(mutex);
}
```

- Only 1 philosopher eating at 1 time
- Avoids deadlocks, but does not maximize concurrency

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 );
    }
}
```



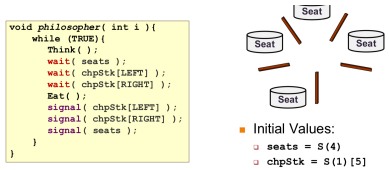

```
void takeChopsticks(i) {
    wait(mutex);
    state[i] = HUNGRY;
    safeToEat(i);
```

```
    signal(mutex);
    wait(s[i]);
}

void safeToEat(i) {
    if ((state[i] == HUNGRY) &&
        ((state[LEFT] != EATING) && (state[RIGHT] != EATING))) {
        state[i] = EATING;
        signal(s[i]);
    }
}

void putChopstick(i) {
    wait(mutex);
    state[i] = THINKING;
    safeToEat(LEFT);
    safeToEat(RIGHT);
    signal(mutex);
}
```

- `mutex`: For blocking critical section of updating state
- `S[i]` semaphore: Blocks philosopher i if hungry and cannot eat
  - All initialized to 0
- Intuition:
  1. If any philosopher next to me is eating, block myself
     1. When a philosopher finishes eating, they signal to me that I have a chance to eat
  2. Else, eat!
     1. Once finish eating, signal to hungry philosophers next to me to eat

Limited Eater

- Still have 5 philosophers



- Intuition: If at most 4 philosophers can sit at the table → There exists empty seat(s) → Attempt 1 works, since no deadlock
  - `seats` semaphore: Blocks if no more seat
    - 5th philosopher can eat after another philosopher gives up seat

Synchronization Implementations

POSIX Semaphore

- Implementation of semaphore under Unix

```
#include<semaphore.h>

gcc something.c -lrt // real time library

// Usage
// Initialize a semaphore
// Perform wait() or signal() on semaphore
```

`pthread` Mutex and Conditional Variables

- For `pthreads`
- Mutex (`pthread_mutex`)
  - Binary semaphore
  - Lock: `pthread_mutex_lock()`
  - Unlock: `pthread_mutex_unlock()`