CS2106 Intro Op. Systems Notes

AY23/24 Sem 2, github.com/gerteck

1. Introduction

Course objectives: Introduces basic concepts in operating systems.
Focusing on:
- OS structure and architecture, process management, memory management, file management and OS protection mechanism.
- Identify and understand major functionalities of modern operating systems.
- Extend and apply the knowledge in future courses.

Supplementary Text: Modern Operating System (5th Edition), by Andrew S. Tanenbaum, Pearson, 2023.

Learning Outcomes

- Understand how an **OS manages computational resources for multiple users and applications, and the impact on application performance**
- Appreciate the **abstractions and interfaces provided by OS**
- Write **multi-process / thread programs** and avoid common pitfalls such as **deadlocks, starvation and race conditions.**
- Write system programs that utilizes **POSIX** syscall for process, memory and I/O management.
- Self-learn and explore advanced OS topics.
- Understand important design principles in complex systems.

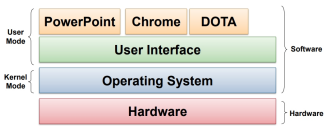
Areas to focus on: Try to understand how things are running in parallel, since we naturally think sequentially. Secondly, how we can manage memory and how they combine and interact (in strange ways), synchronization.

Operating System OS

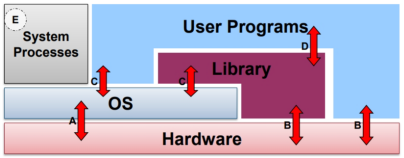
An OS is a program that acts as an intermediary between a computer user and the computer hardware. Motivation for OS:

- Manage resources and coordination. (Resource Allocator: Process synchronization, resource sharing)
- Simplify programming (Abstraction of hardware / hardware virtualization, convenient services)
- Enforce usage policies
- Security and protection
- User Program Portability (across different hardware)
- Efficiency (Optimize for particular usage and hardware).

Kernel Mode: Complete access to all hardware resources.
User Mode: Limited / Controlled access to hardware resources.



Generic OS Components



- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

- OS is known as the **kernel**.
  Program that deals with hardware issues, provide system call interface and special code for interrupt handlers, device drivers.
- Kernel code is different from normal programs:
  No use of system call in kernel code, can't use normal libraries, no normal I/O (must do I/O itself).
- **Implementing OS**: Historically in assembly/machine, now in HLLs (C, C++). Heavily hardware architecture dependent. Challenges include complexity, debugging, codebase size.

OS Structures

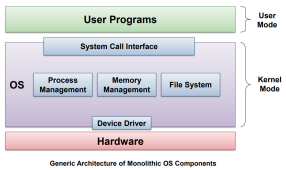
Monolithic OS: One Big program.

- Well understood, good performance, but highly coupled components (everything running in kernel mode) and usually devolved into very complicated internal structure.

Microkernel OS:

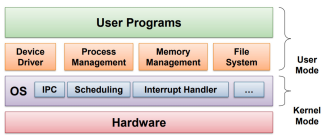
- Kernel is very small and clean, only providing basic and essential facilities.
- Inter-Process Communication (**IPC**), Address space management, Thread management etc.
- Higher level services are built on top of basic facilities, run as server process *outside* of OS, use IPC to communicate.
- Kernel is more robust and extendible, better isolation and protection between kernel and high level services. But, lower performance. (Latency)

Monolithic Kernel Illustration



Generic Architecture of Monolithic OS Components

Microkernel Components



Generic Architecture of Microkernel OS Components

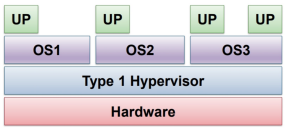
Other OS Structure

- **Layered Systems**: Generalization of monolithic system, organize components into hierarchy of layers. Lowest is hardware, highest is user interface.
- **Client-Server Model**: Variation of microkernel. Two classes of processes: Client p. request service from server process, server process built on top of microkernel. Client & Server process can be on separate machine.

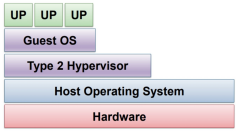
Virtual Machines

- **Motivation**: OS assumes total control of hardware, making it hard to run several OS on same hardware at same time. OS is also hard to debug / monitor, hard to observe working of OS, test potentially destructive implementation.
- **Virtual Machine**: Software emulation of hardware.
- **Virtualization of underlying hardware**: Illusion of complete hardware to level above. (Memory, CPU etc.) Normal OS can then run on top of virtual machine. Aka **Hypervisor**.
  - **Type 1 Hypervisor**: Provides individual virtual machines to guest OSes (e.g. IBM VM/370)
  - **Type 2 Hypervisor**: Runs in host OS, Guest OS runs inside Virtual Machine, (e.g. VMware)

Type 1 Hypervisor



Type 2 Hypervisor



- Upcoming Topics -

OS Process Management: As OS (to maximise efficiency hardware resources), to be able to switch from running one program to the other (share hardware, e.g. CPU), requires information regarding execution of A stored, and A's information replaced with B's information to run. (E.g. the registers in CPU replaced)

- **2. Process Abstraction**: Info describing executing program
- **3. Process Scheduling**: Deciding which process gets to execute
- **4. Inter-Process Communication**: Passing information between processes (tough)
- **5. Threads + Synchronization**: Alternative to Process (Light-weight process aka Thread)
- **6. Memory Management**
- **7. Disjoint Memory Management**
- **8. File System Management**
- **9. File System Implementation**

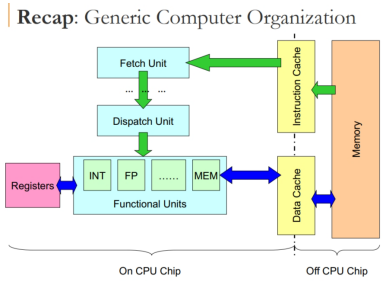
2. Process Abstraction

To switch programs, requires information of both programs. Hence, we need abstraction to describe running program, aka **process**.

- **(Process / Task / Job)** is a dynamic abstraction for executing program.
- It is info required to describe a *running program*:
  - Memory Context (Code/Text, Data, Stack, Heap),
  - Hardware Context (Register/PC, Stack/Frame Pointer),
  - OS Context (Process Properties (PID, State), Resources Used).

Computer Organization (Recap)

- **Components**: Memory, Cache, Fetch Unit (Loads instruction, location indicated by special register **PC**.)
- **Functional Units** (Carry out instruction execution, dedicated to diff. instr. type) (CS2100 looked at INT func. unit)
- Registers (Internal storage, fastest access speed).
  - **GPR**: General Purpose Register, accessible by user program / compiler.
  - **Special Registers**: PC, Stack/Frame Pointer, PSW etc.
- **Binary Executable File**: file in machine language (built by compiler) for specific processor:
  - Executable (binary) consists two major components: Instr. (Text) & Data
  - When under execution, more info: Memory, Hardware, OS context.



Memory Context for Function Call (Stack Memory)

Memory Context Challenges of Functional Calls:

- Control Flow Issues: Need to jump to function body, resume after, need to store PC of caller.
- Data Storage Issues: Need to pass params to function, capture return result, may need declare local variables.
- Require region of memory dynamically used by function invocations.

Hence, portion of memory space used as **stack memory** that stores executing function using **stack frame**, which includes usage of *Stack Pointer, Frame Pointer*.

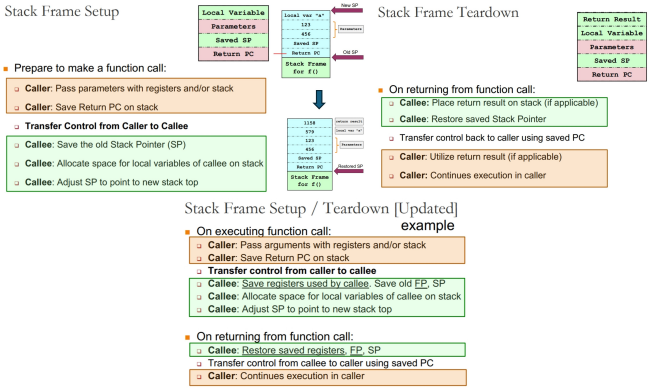
Stack Memory Region

- **Memory region to store information function invocation.**
- **Stack Frame**: Describes information of function invocation.
- Stack frame added on top when function is invoked, stack "grows", removed from top when function call ends, stack "shrinks".
- Stack Frame contains return PC address of caller, arguments for function, storage for local variable, etc.
- **Stack Pointer**: Indicates top of stack region (first unused memory location). Usually indicated in specialized register.

Function Call Convention: Stack Frame Setup / Teardown

There are different ways to setup stack frame, known as function call convention, differences about (info stored in frame, which portion of stack frame prepared & cleared by caller / callee etc). Dependent on hardware & programming language.

Example Scheme:



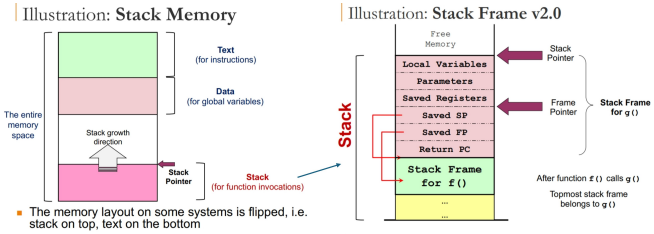
Stack Frame: Other Information

Frame Pointer

- To facilitate access of various stack frame items. As stack pointer hard to use as it can change, some processors provide dedicated register Frame Pointer.
- Frame Pointer points to fixed location in stack frame, other items accessed as displacement from frame pointer, usage of FP is platform dependent.

Saved Registers

- Since number of GPR limited, when GPR exhausted, use memory to temp. hold GPR values for reuse.
- **Known as Register Spilling**. Function can spill registers it intends to use before function starts, then restore registers at end of function.



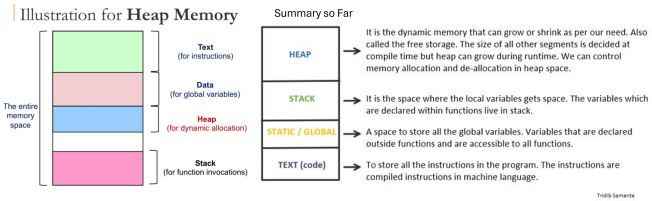
Memory Context for Dynamically Allocated Mem. (Heap)

Most programming languages allow dynamically allocated memory, i.e. acquire memory space during execution time.

- In C, `malloc()` function call, while C++ / Java: `new` keyword.
- Cannot place in *Data region*, as allocated at runtime, size not known during compilation.
- Cannot place in *Stack region*, as no definite deallocation time, cannot be freed by garbage collector.
- Solution: Set up separate **Heap Memory Region**

Heap Memory

- Managing heap memory trickier due to variable size, variable allocation / deallocation timing.
- Common situation where heap memory alloc/dealloc creating "holes" in memory. Free memory block squeezed between occupied memory blocks.
- Covered in memory management.



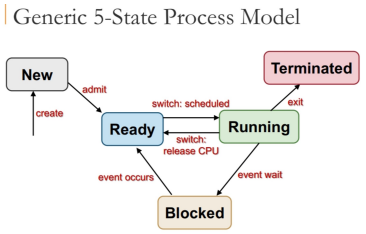
OS Context: Process ID, Process State

Process Identification:

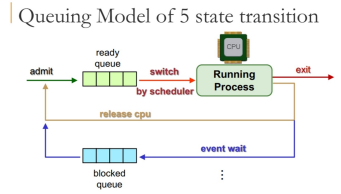
- **Process ID**: To distinguish processes from each other (Just a number, unique among processes)
- PIDs are OS dependent as well, including if PIDs reused, if limits maximum no. of processes or any PIDs reserved.

Process State:

- Processes require a process state as indication of execution status. (Running / Not Running / Ready to Run etc.)
- **Process Model**: Set of states and transitions, describes behaviors of a process.
- **Global View of Process States**: Given n processes,
  - With 1 CPU, ≤ 1 process in running state, 1 transition at a time.
  - With m CPUs, $\leq$ m processes running state, possibly parallel transitions.
- Different processes may be in different states, each process may be in different part of its state diagram.
- **5-State Process Model**:



Notes: generic process states, details vary in actual OS



Notes:
- More than 1 process can be in ready + blocked queues
- May have separate event queues
- Queuing model gives global view of the processes, i.e. how the OS views them

Process States for 5-Stage Model

- **New**:
  - New process created
  - May still be under initialization → not yet ready
- **Ready**:
  - process is waiting to run
- **Running**:
  - Process being executed on CPU
- **Blocked**:
  - Process waiting (sleeping) for event
  - Cannot execute until event is available
- **Terminated**:
  - Process has finished execution, may require OS cleanup

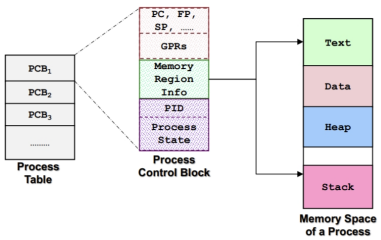
Process State Transitions in 5-Stage Model

- **Create** (nil → New):
  - New process is created
- **Admit** (New → Ready):
  - Process ready to be scheduled for running
- **Switch** (Ready → Running):
  - Process selected to run
- **Switch** (Running → Ready):
  - Process gives up CPU voluntarily or *preempted* by scheduler
- **Event wait** (Running → Blocked):
  - Process requests event/resource/service which is not available/in progress
  - Example events:
    - System call, waiting for I/O, *(more later)*
- **Event occurs** (Blocked → Ready):
  - Event occurs → process can continue

Process Table & Process Control Block

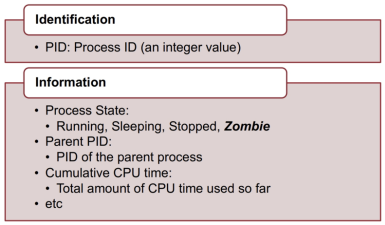
- Since the OS is just a program as well, need to make use of data structures to track these processes.
- **Process Control Block (PCB) or Process Table Entry**: Entire Execution Context for a process.
- Kernel maintains **PCB** for all processes. (Conceptually stored as one table representing all processes.)
- **Factors to consider**:
  - Scalability (how many concurrent processes at once).
  - Efficiency (should provide efficient access with minimum space wastage).

Illustration of a Process Table



Process Abstraction in Unix

- **Process Identification, Information**



- Unix Command for process information:
  - `ps` (short for process status)

- **Process Creation, Termination, Parent-Child Synchronization**

Note: Command Line Argument in C

- We can pass arguments to a program in C.
- `argc` : Number of CL arguments, including program name.
- `argv` : A char strings array, each element in `argv[]` is a C character string.

```
int main( int argc, char* argv[] )
{ int i;
   for (i = 0; i < argc; i++){
   printf("Arg \%i: \%s, ",i, argv[i] );
   }
   return 0;
}
```

- Example Run: "a.out 123 hello world"
- Output: "Arg 0: a.out, Arg 1: 123, Arg 2: hello, Arg 3: world"

Process Creation: `fork()`

Process Creation in Unix: `fork()`

- The main way to create a new process

Header File:
```
#include <unistd.h>
```

Syntax:
```
int fork( );
```

- Returns:
  - PID of the newly created process (for parent process) OR
  - 0 (for child process)
- Header files are system dependent
  - *"man fork"* to locate the right files for your system!

Process Creation in Unix: `fork()` (cont)

- Behavior:
  - Creates a new process (known as ***child process***)
  - Child process is a **duplicate** of the current executable image
    - i.e. same code, same address space etc
    - Data in child is a **COPY** of the parent (ie.not shared)
  - Child **differs only in:**
    - Process id (PID)
    - Parent (PPID)
      - Parent = The process which executed the fork()
    - `fork()` return value

`fork()` : Example

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

int main()
{
    printf("I am ONE\n");
    fork();
    printf("I am seeing DOUBLE\n");

    return 0;
}
```

- Question:
  - What do you think is the output?

- `fork()` : Create exact copy of the parent, including any variables.
- **Output**: Both parent and child resume execution after the point `fork()` .
- **Note** `clone()` : `fork()` not versatile, for scenarios where partial duplication preferred, `clone()` , which supersedes `fork()` .
- Both parent and child processes continue executing, common usage is to use the parent/child process differently. (Parent spawn off child to carry out some work, parent ready to take another order.)
  - Use return value of `fork()` to distinguish parent and child.

`fork()`: Parent and Child Example

```
... ...
int result;

result = fork();
if (result != 0){
    printf("P:My Id is %i\n",getpid());
    printf("P:Child Id is %i\n",result);
} else {
    printf("C:My Id is %i\n", getpid() );
    printf("C:Parent Id is %i\n", getppid() );
}
... ...
```

Parent Process | Using fork() result to differentiate parent and child | Child Process

Process Replacement: `execl()` System Call

- Function **replaces current executing process image** with a new process image specified by path. No return is made because the calling process image is replaced by the new process image.
- **Code Replacement**, but **PID and other information still intact**.

`execl()` System Call

Header File:
```
#include <unistd.h>
```

Syntax:
```
int execl( const char *path,
           const char *arg0,
           ...,
           const char *argN, NULL );
```

- `path`: Location of the executable
- `arg0`, ..., `argN`: Command Line Argument(s)
- `NULL`: To indicate end of argument list

`execl()`: Simple Example

```
int main()
{
    execl( "/bin/ls", "ls", "-l", NULL);
}
```

- Note:
  - `Path` = "`bin/ls`"
    - The "`dir`" command in unix, to list the files in directory
  - `arg0` = "`ls`"
    - The program name
  - `argl` = "`-l`"
- The above is exactly the same as executing: `ls -l`

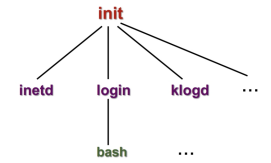
- **By combining** `fork()` **and** `exec()` , we can spawn off a child process (let it perform task through `exec()`), while parent process around to accept another request.
- This combination of mechanisms is main way in Unix to get new process for running new program!

The Master Process: `init`

- Every process has parent process, consider special initial process.
- `init` **process**: Created in kernel at boot up time, usually PID = 1.
- **Purpose**: Watches and respawns other (critical) processes where needed.
- `fork()` creates the process tree, where `init` is the root process.

Simplified Process Tree Ex.

Process Tree Example (simplified)



Note: just a simple example, actual process tree varies according to Unix setup

- **d**, (e.g. `klogd`) at end of process name usually means server process (Daemon, background process.)

Process Termination in Unix: `exit`

Process Termination in Unix

- To end execution of process:

Header File:
```
#include <stdlib.h>
```

Syntax:
```
void exit( int status );
```

- Status is returned to the parent process (more later)
- Unix Convention:
  - 0 = Normal Termination (successful execution)
  - ! 0 = To indicate problematic execution
- The function **does not return!**

Implicit `exit()`

- Most programs have no explicit exit() call
- Example:

```
int main()
{
    printf("Just to say goodbye!\n");
}
```

- Return from `main()` implicitly calls exit()
  - Open files also get flushed automatically!

- **Process finished execution**: **Most** system resources used by process are released on exit. (e.g. file descriptors).
- **Certain basic process resources not Releasable**: PID, status needed. For parent-children synchronization, for parent to check status of child, For process accounting info (e.g. cpu time).
- Process table entry may still be needed after termination.

Parent/Child Synchronization in Unix

- Parent process can wait for child process to terminate.
- Argument is a pointer to a variable that will store the return value (`*status`).

Parent/Child Synchronization in Unix

Header File:
```
#include <sys/types.h>
#include <sys/wait.h>
```

Syntax:
```
int wait( int *status );
```

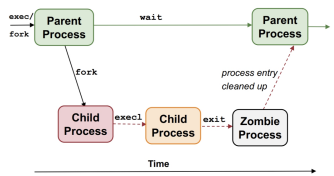
- Returns the PID of the terminated child process
- status (passed by address):
  - Stores the exit status of the terminated child process
  - Use **NULL** if you do not need/want this info

Parent/Child Synchronization in Unix

- Behavior:
  - The call is blocking:
    - Parent process blocks until at least one child terminates
  - The call cleans up ***remainder*** of child system resources
    - Those not removed on `exit()`
    - Kill zombie process ☺
- Other variants of `wait()` :
  - `waitpid()`
    - Wait for a specific child process
  - `waitid()`
    - Wait for any child process to **change status**
  - etc…

- Kills zombine processes! With enough zombies, process table finite size, run out of space.

Process Interaction in Unix



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

Zombie Processes (2 Cases)

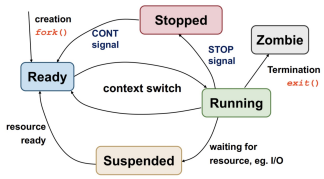
- `wait()` "creates" the zombies (and later cleans it up) as on process exit, process becomes zombie.
- Since it cannot delete all process info (if parent asks for info in `wait()` call, remainder of process data structure can be cleaned up only when `wait()` happens.)
- We cannot `kill PID` zombie process, is already dead. Until restart system or modern OS look through table and remove them.

1. **Parent process terminates before child process**
   - `init` process becomes "pseudo" parent of child processes.
   - Child termination sends signal to `init`, which utilizes `wait()` to cleanup
2. **Child process terminates before parent but parent did not call wait**
   - Child process become a zombie process
   - Can fill up / hog process table, May need a reboot to clear the table on older Unix implementations

Summary of Unix Process System calls

- `fork()`:
  - Process creation
- `exec()` family:
  - Change executing image/program
  - `execl, execv, execve, execle, execvp`
- `exit()`:
  - Process termination
- `wait()` family:
  - Get exit status, synchronize with child
  - `wait, waitpid, waitid, etc`
- `getpid()` family:
  - Get process information
  - `getpid, getppid`, etc

Process State Diagram in Unix



Implementation Issues

Implementing `fork()`

Implementing `fork()`

- Behavior of `fork()`:
  - Makes an almost exact copy of parent process
- Simplified implementation:
  1. Create address space of child process
  2. Allocate `p'` = new `PID`
  3. Create kernel process data structures
     - E.g. Entry in Process Table
  4. Copy kernel environment of parent process
     - E.g. Priority (for process scheduling)
  5. Initialize child process context:
     - `PID=p', PPID=parent` id, zero CPU time

Implementing `fork()` (cont)

6. Copy memory regions from parent
   - Program, Data, Stack
   - Very expensive operation that can be optimized (more later)
7. Acquires shared resources:
   - Open files, current working directory etc
8. Initialize hardware context for child process:
   - Copy registers, etc. from parent process
9. Child process is now ready to run
   - add to scheduler queue

Memory copy is very expensive:

- Potentially need to copy the whole memory space

- **Copying entire memory space is wasteful** and not always needed! (E.g. copy entire 200mb program image of Zoom etc). Mostly, only need contents, and PC, register values.
- **Give Rise to COW.** (copy on write)

Memory Copy Operation

- If child just read from location, unchanged, just use a shared version.
- **Only when write is perform on a location, then two independent copies needed.**
- **Copy on Write** is possible optimization, only duplicate "memory location" when it is written to, otherwise parent and child share same "memory location".
- Note: memory organized into memory pages (consec range of mem locations), memory managed on page level instead of individual location.

System Calls (Process Interaction with OS)

API to OS: Application Program Interface to OS

- OS API provides way of calling facilities/services in kernel.
- **Not same as normal function call**: Change from *user mode to kernel mode*.
- Different OS have different APIs: Unix Variants most follow POSIX standards, small no. of calls 100. Windows family uses Win API across diff. windows, huge no. of calls 1000.

Unix System Calls in C/C++ program

- In C/C++ program, **system call can be invoked almost directly**, as library version very closely reflects these calls.
- Majority of system calls have library version with **same name** and parameters, library version acts as **function wrapper**.
- A few library functions present more user friendly version, e.g. less no. /more flexible parameters). Library version acts as **function adapter**.

Example

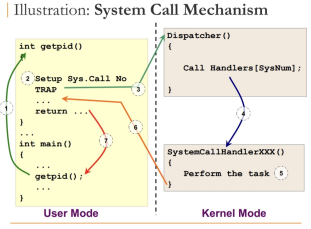
```
#include <unistd.h>
#include <stdio.h>

int main()
{
    int pid;

    /* get Process ID */
    pid = getpid();

    printf("process id = %d\n", pid);

    return 0;
}
```

Library call that has the same name as a system call

Library call that make a system call

- System Calls invoked in this example:
  - `getpid()`
  - `write()` – made by `printf()` library call

Illustration: System Call Mechanism



General System Call Mechanism

General System Call Mechanism

1. User program invokes the library call
   - Using the normal function call mechanism as discussed
2. Library call (usually in assembly code) places the **system call number** in a designated location
   - E.g. Register
3. Library call executes a special instruction to switch from user mode to kernel mode
   - That instruction is commonly known as **TRAP**

General System Call Mechanism (cont)

4. Now in kernel mode, the appropriate system call handler is determined:
   - Using the system call number as index
   - This step is usually handled by a **dispatcher**
5. System call handler is executed:
   - Carry out the actual request
6. System call handler ended:
   - Control return to the library call
   - Switch from kernel mode to user mode
7. Library call return to the user program:
   - via normal function return mechanism

Exception and Interrupt

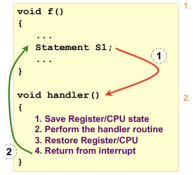
Exception

- **Executing machine level instruction** can cause **exception**. For example:
  - Airthmetic Errors: Overflow, Underflow, Division by Zero
  - Memory Accessing Errors: (accessing memory not belonging to program), Illegal memory address, mis-aligned memory access.
- Exception is **Synchronous**: Determinate, occurs due to program execution at exact points of time.
- **Effect of Exception**: Have to execute **exception handler**, which is similar to a **"forced function call"!**
- Exception ≠ Interrupt!

Interrupt

- **External events** can interrupt the execution of a program.
- Interrupt request lines connected to CPU, lines are checked during instruction execution cycle.
- Usually hardware related, e.g.: Timer, Mouse move, Keyboard press etc
- Interrupt is **asynchronous**: Events occurs independent of program execution.
- **Effect of interrupt:** Program execution, suspended, execute an interrupt handler.

Exception/Interrupt Handler: Illustration



1. Exception/Interrupt occurs:
   - Control transfer to a handler routine **automatically**
2. Return from handler routine:
   - Program execution resume
   - **May** behave as if nothing happened

Summary

- Using **process as an abstraction** of running program.
- Includes necessary information (environment) of execution, Memory, Hardware and OS contexts.
- **Process from OS perspective**: PCB and process table
- **How OS & Process interact**: System calls, Exception / Interrupt

REFER TO TEXTBOOK:

- **Modern Operating System (3rd Edition)**
  - Section 2.1: Processes
  - Section 2.4: Process Scheduling
  - Section 2.2: Threads
- **Operating System Concepts (8th Edition)**
- Section 3.1

3. Process Scheduling

A multiprogrammed computer frequently has multiple processes/threads computing for CPU at the same time. Occurs whenever ≥ 2 simultaneously in ready state.

- **Scheduler**: Part of OS to decide which process to run next.
- **Scheduling Algorithm**: Algo used.
- **Scheduling Problem**: Choosing, ready process $>$ available CPUs.
- In addition to picking right process, need **efficient use of CPU** as process switching is expensive. (Switch user to kernel mode, save state of process, memory map, memory cache may need to reload, etc.)
- **I/O Input/Output** (disk or network): When process enters blocked state waiting for external device to complete work.

Process Behavior

- Process' unique **requirement of CPU time.**
- Process goes through phases of CPU-activity & IO-activity.
- **Compute/CPU-Bound Process** (computation, e.g. number crunching) vs. **IO-bound Process** (e.g. read/write to file, print to screen)

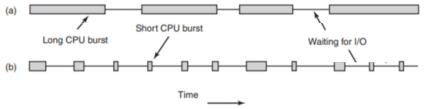


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Process Environment

- **Batch Processing**: No user, no interaction / responsiveness required.
- **Interactive / Multiprogramming**: Active user interacting with system. Need responsive, consistent in response time.
- **Real Time Processing**: Deadline to meet, usually periodic process.

Scheduler Evaluation Criteria

- Many criteria to evaluating algo, largely influenced by p. environment. May be conflicting.
- **All Systems**:
  - *Fairness*: CPU time (per process basis / per user basis). **No starvation**.
  - *Balance*: All parts of computing system utilized.
- **Batch Systems**:
  - *Throughput*: Maximize jobs per hour.
  - *Turnaround time*: Minimize time btwn. submission & termination.
  - *++(Waiting Time)*: Related to turnaround, time waiting for CPU
  - *CPU utilization*: keep CPU busy all the time.
- **Interactive Systems**:
  - *Response time*: respond to requests quickly.
  - *Proportionality*: meet users' expectations.
- **Real-time Systems**:
  - *Meeting deadlines*: avoid losing data. (e.g. livestream)
  - *Predictability*: avoid quality degradation in multimedia.

Concurrent Execution

- **Concurrent Processes**: Logical concept to cover multitasked processes.
- **Virtual parallelism**: Illusion of parallelism (pseudo-parallelism)
- **Physical parallelism**: Multiple CPUs/Cores, multiple parallel exec

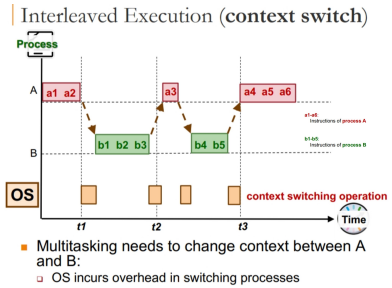
When to Schedule

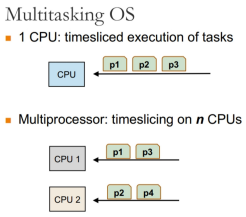
Key issue, when to make schedule decisions.
E.g. When new process created, run parent or child. When process exits, when process blocks on I/O, or I/O interrupt.

- **Non-preemptive (cooperative)**: Process stays scheduled (running state) until it blocks, or gives up CPU voluntarily.
- **Preemptive (Fixed Quota)**: Process given fixed time quota to run (possible to block / give up early). At end of quota, running process suspended, another picked if available.

Interleaved Execution (Timeslicing)

- **Concurrent Execution on 1 CPU**: Interleave instructions from both processes.
- **OS overhead**: Multitasking needs to change context between programs, incurs overhead.





- **Scheduling a Process**:
  1. Scheduler triggered (OS takes over)
  2. If Context Switch needed, save context, place on blocked/ready queue.
  3. Pick suitable process P to run base on scheduling algo.
  4. Setup context for P.
  5. Let process P run.

Scheduling in Batch Systems

Environment: No user interaction, non-preemptive scheduling predominant.
Scheduling algoritms generally easier to understand and implement, with variants and improvements for other type of system.
Criteria: Turnaround time (related to waiting time, time spent waiting for CPU). Throughput. CPU utilization.

Batch Systems Scheduling Algorithms

FCFS: First-Come First-Served

- Tasks stored on **FIFO queue based on arrival time**.
- Pick first task in queue to run until done / blocked. Blocked task removed from FIFO queue, when ready, place at back of queue ("newly arrived").
- **Evaluation**:
  - *No starvation*. (Every task eventually processed)
  - *Covoy Effect*. (CPU-Bound followed by IO-Bound tasks heavily inefficient.) Simple reordering can reduce average waiting time.

SJF: Shortest Job First (Nonpreemptive)

- Select task with **smallest total CPU time**.
- Need to know total CPU time for task in advance.
  - Possible to guess future CPU time by previous CPU-bound phases.

Common approach (Exponential Average):

$$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$$

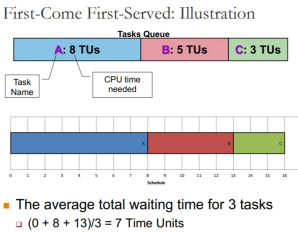
- $Actual_n$ = The most recent CPU time consumed
- $Predicted_n$ = The past history of CPU Time consumed
- α = Weight placed on recent event or past history
- $Predicted_{n+1}$ = Latest prediction

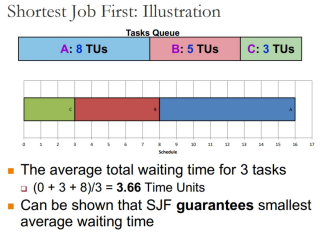
- **Evaluation**:
  - *Starvation Possible* (Biased towards short jobs) - *Minimize average waiting time*. - Optimal only when all jobs available simultaneously.

SRT: Shortest Remaining Time Next (Preemptive)

- **Preemptive ver of SJF**.
  - Scheduler chooses process whose remaining run time is shortest.
- When new job arrives, total time compared to current process' remaining (or expected) time.

Batch System Scheduling Examples

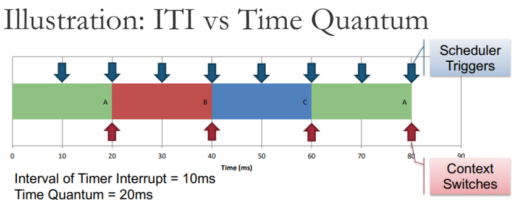




Scheduling in Interactive Systems

Criteria: Response time (time btwn request & response). Predictability (Less variation in response time).
Environment: User interaction. Preemptive scheduling used to ensure good response time. Scheduler needs to run periodically.
Ensuring Periodic Scheduler: Use timer **interrupts** (based on hardware clock). OS ensures timer interrupt cannot be intercepted by any other program. Interrupt handle **invokes scheduler**.

- **ITI: Interval of Timer Interrupt**: Timing Interval that interrupt happens, and OS scheduler triggered. Typical values (1ms - 10ms).
- **Time Quantum**: Execution duration given to each process, constant / variable. Must be multiples of ITI. Typical values (5ms - 100ms).



Interactive Systems Scheduling Algorithms

RR: Round Robin

- Preemptive ver. of FCFS.
- **Tasks stored in FIFO queue**. Each process assigned time quantum.
- If task still running at end of quantum / blocked / gives up CPU voluntarily, CPU preempted, given to another process.
- Task placed at end of queue to wait for another turn.
- **Response time guarantee**: n tasks, q quantum.
  Time before task gets CPU bounded by $(n-1)q$.
- **Evaluation**:
  - Too short quantum = many process switches, lower CPU efficiency
  - Too long quantum causes poor response to short interactive requests.

Priority Scheduling: Priority Based

- Each process assigned a priority, runnable process with highest priority p allowed to run.
  - **Preemptive ver.**: Higher p. process can preempt running low p. process.
  - **Non-preemptive ver.**: Late high p. process wait for next scheduling.
- **Evaluation**:
  - **Possible Starvation**: High p. process may hog CPU.
  To prevent this, scheduler may **decrease priority** of currently running process at each clock tick (clock interrupt). Or, give some max time quantum, when used up, allow next in line to run.
  - **Priority Inversion**: When lower priority task preempts higher priority task. (If lower priority task locks some resource, e.g. file, gets switched, higher priority task cannot run)

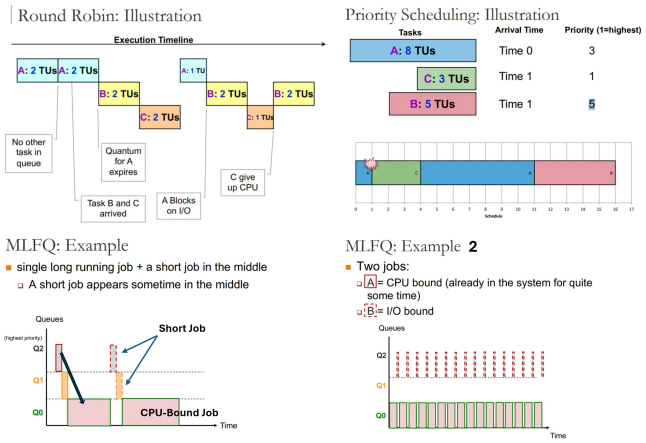
MLFQ: Multi-Level Feedback Queue

- More efficient to give CPU-bound process large quantum once in a while, rather than small quanta frequently to reduce swapping. Also, giving all processes large quantum means poor response time.
  Hence, solution to set **multiple priority queues**.
- As (CPU-bound) process sinks deeper into priority queues, run less frequently, saving CPU for short, interactive processes.
- Hence, **adaptively learn process behavior**, minimize both:
  - Min. Response time for IO bound processes
  - Min. Turnaround time for CPU bound processes.
- **MLFQ Rules**:
  - **Basic Rule**: Higher priority process runs. If same p, run in RR.
  - **Priority Setting**: New job given highest priority. If job fully utilize time slice, priority reduced. If job gives up / blocks before time slice finished, priority retained.
- **Evaluation**: - **Can be gamed**: User typing carriage returns at random every few seconds doing wonders for his response time.

Lottery Scheduling

- Give processes lottery tickets for system resources. When scheduling, chose ticket at random.
- "All processes equal, some processes more equal." More important processes given extra tickets.
- **Evaluation**:
  - *Responsive*: New process can participate in next lottery.
  - *Good Control*: Each process can distribute to child processes proportionally w.r.t to need.

Interactive System Scheduling Examples



Others

- **Shortest Process Next**: Estimate (using calculated aging).
- **Guaranteed / Fair-Share Scheduling**: Track CPU usage, run accordingly. Each user gets agreed allocation of CPU.

4. Threads

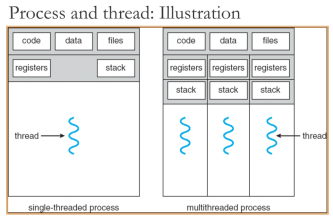
In trad. OS, each process has an address space and single thread of control. Desirable instead to have multiple threads of control in the same address space running in quasi-prallel, as though (almost) separate processes (except for shared address space).

Motivation for Thread

- **Process is Expensive**: Process creation under `fork()` model duplicate memory space, most of process context.
- Context switch also requires overhead, save/restore process info.
- **Communication**: Hard for independent processes to communicate with each other. Independent memory space, no shared variable, requires IPC.
- **Thread**: "quick hack" into popular mechanism.
- **Basic Idea**: Traditional process only single thread of control. (Only one instruction executing at a time).
- Add more threads (of control) to same program, multiply parts of programs executing at same time conceptually.

Process and Thread

- **Multi-threaded Process**: Single process can have multiple threads.
- Threads in same process share same: *Memory Context* (text, data heap), *OS Context* (Process id, files etc.)
- **Unique info per thread**: Id (thread id), Registers (GPR and special), "Stack".
- **Process Context Switch vs. Thread Switch**:
  - *Process Context Switch*: Switch OS, Hardware, Memory Context.
  - *Thread Switch (same process)*: Just hardware context (registers, stack).
  - Thread is **lightweight process**.



Threads: Benefits

- **Resource Sharing**: Ability for parallel entities to share address space and all of data. No need add. mechanism for passing info.
- **Economy**: Threads are lighter weight than process, faster to create and destroy. Less resources to manage.
- **Responsiveness**: No performance gain when all CPU bound, but when substantial computing and substantial I/O, threads allows activities to overlap, speeding up application.
- **Scalability**: Multithreaded program can take advantage of multiple CPUs.

Threads: Problems

- **System Call Concurrency**: Parallel execution of multiple threads: parallel system call possible. Need to guarantee correctness and determine correct behavior.
- **Process Behavior (OS dependent)**: Impact on process operations, (e.g. If one thread executes exec() / exit(), what about other threads / whole process).

Thread Models (ways to support threads)

- **User Thread**: Thread implemented as a **user library**. (Runtime system (in the process) will handle thread related operation.)
- Kernel not aware of threads in process.
- **Kernel Thread**: Thread implemented in the OS. Operation handled as system calls.
- Kernel thread-level scheduling possible, where kernel schedule by threads instead of by process.
- Kernel may make use of threads for own execution.
- **Threads on Modern Processor**: Threads started as software mechanism, now exists hardware support on modern processors (multiple register sets on same core, *simultaneous multi-threading (SMT)*, "Hyperthreading" on Intel).

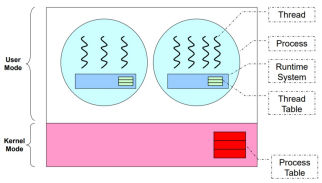
User Thread

- **Advantages**: Can have any multithreaded program on any OS, thread operations are just library calls, generally more configurable and flexible (e.g. customized thread scheduling policy.)
- **Disadvantages**: OS not aware of threads, scheduling is performed at process level. (One thread blocked, process blocked, all threads blocked). Cannot exploit multiple CPUs.

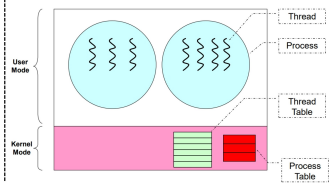
Kernel Thread

- **Advantages**: Kernel can schedule on thread levels: >1 thread in same process can run simultaneouly on multiple CPUs.
- **Disadvantages**: Thread operations now system calls, slower and more resource intensive. Generally less flexible, used by all multithreaded programs, so too many / few features, overkill / not flexible enough for different programs.

User Thread: Illustration



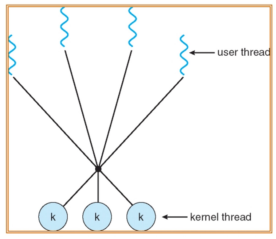
Kernel Thread: Illustration



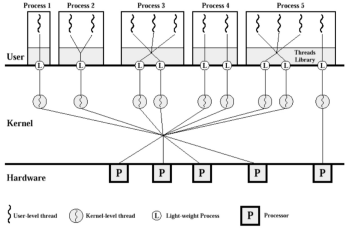
Hybrid Thread Model

- Have both Kernel and User threads, where OS schedule on kernel threads only, and user thread binds to a kernel thread.
- Offers flexibility, limit concurrency of any process / user.

Hybrid Thread Model



Hybrid Model Example: **Solaris**



POSIX Threads

IEEE defined standards (1003.1c) for threads, called *Pthreads*, most UNIX systems support it. Defines over 60 function calls.

Pthread

- **POSIX**: Portable Operating System Interface. (family of standards)
- Defines the API as well as the behavior, but not implementation.
- pthread can be implemented as user / kernel thread.

Basics of pthread

- Header File:
  - `#include <pthread.h>`
- Compilation (flag is system dependent):
  - `gcc xxxx.c -lpthread`
- Useful datatypes:
  - `pthread_t` : Data type to represent a thread id (TID)
  - `pthread_attr`: Data type to represents attributes of a thread

pthread Creation Syntax

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

- Returns (0 = success; !0 = errors)
- Parameters:
  - `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 Termination Syntax

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

- Parameters:
  - `exitValue`: Value to be returned to whoever synchronize with this thread (more later)
- If `pthread_exit()` is not used, a *pthread* will terminate automatically at the end of the `startRoutine`
  - If a "return XYZ;" statement is used, then "XYZ" is captured as the exitValue
  - Otherwise, the exitValue is not well defined

pthread Creation & Termination: Example

```
//header files not shown

void* sayHello( void* arg )          Function to be executed by a pthread
{
    printf("Just to say hello!\n");
    pthread_exit( NULL );            Pthread Termination
}

int main()
{
    pthread_t tid;                   Pthread Creation

    pthread_create( &tid, NULL, sayHello, NULL );
    printf("Thread created with tid %i\n", tid);

    return 0;
}
```

pthread: Sharing of memory space

```
//header files not shown
int globalVar;          Variable shared between pthreads

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

int main()
    pthread_t tid[5];    //5 threads id
    int i;
    for (i = 0; i < 5; i++)
        pthread_create( &tid[i], NULL, doSum, NULL );
    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

Bc. **Race Condition**, Sum unpredictable

```
#include <stdio.h>
#include <pthread.h>
int globalVar;

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


int main() {
    pthread_t tid[5]; //5 threads id
    int i;

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

    //Wait for all threads to finish
    for (i = 0; i < 5; i++) {
        pthread_join( tid[i], NULL ); }

    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

pthread Simple Synchronization - Join

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

- To wait for the termination of another ***pthread***:
- Returns(0 = success; !0 = errors)
- Parameters:
  - `threadID`: `TID` of the ***pthread*** to wait for
  - `status`: Exit value returned by the target ***pthread***

Pthread: A lot more!

- There are more interesting stuff about ***pthread***:
  - Yielding (giving up CPU voluntarily)
  - Advanced synchronization
  - Scheduling policies
  - Binding to kernel threads
  - Etc
- As we cover new topics, you can explore the ***pthread*** library to see the application!

Summary of Pthread

- **All Pthread threads have certain properties**: each one has identifier, set of registers (including program counter), set of attributes which are stored in a structure.
- Attributes include stack size, scheduling params etc.
- *pthread_create call*: New thread created, returns thread identifier of newly created thread as function value.
- *pthread_exit*: Thread terminate by calling, stops thread and releases stack.
- *pthread_join*: Thread needing to wait for another thread to finish work and exit before continuing can call to wait for specific other thread to terminate. (TID) of thread to wait for given as parameter.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

5. Inter-Process Communication (IPC)

Given that processes frequently need to communicate, some need for communication, preferably in some well-structured way not using interrupts. Consider that cooperating processes have independent memory space, making IPC necessary.

Common Communication Mechanisms

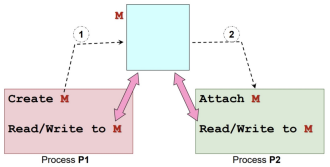
Shared Memory

- **General Idea**: Process 1 creates shared memory region M, process 2 attaches M to its own memory space. Both processes can now communicate using memory region M. Applicable to multiple processes.
- M behaves similar to normal memory region, any writes to region can be seen by all other parties.

POSIX Shared Memory in *nix

- Basic steps of usage:
  1. Create/locate a shared memory region **M**
  2. Attach **M** to process memory space
  3. Read from/Write to **M**
     - Values written visible to all process that share **M**
  4. Detach **M** from memory space after use
  5. Destroy **M**
     - Only one process need to do this
     - Can only destroy if **M** is not attached to any process

Shared-Memory: Illustration



- OS is involved in step 1 and 2 only

- **Advantages**:
  - Efficient, as only initial steps of create & attach M involves OS.
  - Ease of use, as M behaves as normal memory space, info of any size/type writable easily.
- **Disadvantages**:
  - Synchronization: Shared resource, need to sync access still.
  - Implementation is harder.

Example: Master program

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

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

    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);

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

    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]);
    }

    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);

    return 0;
}
```

The master program create the shared memory region and wait for the "slave" program to produce values before proceeding.

Step 1. Create Shared Memory region.

Step 2. Attach Shared Memory region.

The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

The next 3 elements are values produced by the slave program.

Step 4+5. Detach and destroy Shared Memory region.

Example: Slave program

```
//similar header files
int main()
{   int shmid, i, input, *shm;

    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

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

    for (i = 0; i < 3; i++){
        scanf("%d", &input);
        shm[i+1] = input;
    }
    shm[0] = 1;

    shmdt( (char*)shm );
    return 0;
}
```

Step 1. By using the shared memory region id directly, we skip ***shmget***() in this case.

Step 2. Attach to shared memory region.

Write 3 values into shm[1 to 3]

Let master program know we are done!

Step 4. Detach Shared Memory region.

Message Passing

- **General Idea**: Process 1 prepares, send message M. Process 2 receives it.
- Message sending / receiving usually provided as system calls.
- **Additional properties**: Naming (identify other party), synchronization (behavior of send/rec ops).
- **Msg** has to be stored in kernel memory space. Each send/rec op needs to go through OS (system call).

Direct Communication (Naming Scheme)

- Sender/Receiver of message **explicitly names other party**.
- E.g. `Send(P2, Msg)` & `Receive(P1, Msg)`.
- One link per pair of process, need know identity of other party.

Indirect Communication (Naming Scheme)

- Messages sent / recieved from message storage, **aka mailbox / port**.
- E.g. `Send(MB, Msg)` & `Receive(MB, Msg)`.
- One mailbox can be shared among a number of processes

Synchronization Behaviors

- **Blocking Primitives (Synchronous)**: Sender blocked until message received, receiver blocked until message has arrived.
- **Non-Block Primitives (Asynchronous)**: Sender resume operation immediately, receiver either receive, or indicate message not ready yet.

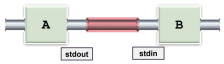
- **Advantages**:
  - Portable, easily implemented on diff. processing env.
  - Easier synchronization: Esp. when synchronous primitive used.
- **Disadvantages**:
  - Inefficient, require OS intervention
  - Harder to use, less flexi, messages limited in size / format.

Pipe (Unix Specific)

- Unix process has 3 default communication channels:
  - **stdin** (standard in): Commonly linked to keyboard input.
  - **stderr** (standard error): Only used print out error msg.
  - **stdout** (standard out): Commonly linked to screen.

Piping in Shell

- For example (" A | B"):



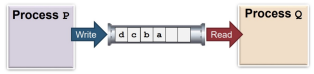
- The output of A (instead of going to screen) directly goes into B as input (as if it come from keyboard)

- General Idea:
  - A communication channel is created with 2 ends:
    - 1 end for reading, the other for writing
    - Just like a water pipe in the real world

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

- **Unix Shell Piping**: Unix shell provides `—` symbol to link input/output channels of one process to another, aka *piping*.

Unix Pipes: as a IPC Mechanism



- A pipe can be shared between two processes
- A form of Producer-Consumer relationship
  - P produces (writes) **n** bytes
  - Q consumes (reads) **m** bytes
- Behavior:
  - Like an anonymous file
  - FIFO ➔ must access data in order

Unix Pipes: Semantic

- Pipe functions as **circular bounded byte buffer** with **implicit synchronization**:
  - Writers **wait** when buffer is **full**
  - Readers **wait** when buffer is **empty**
- Variants:
  - Can have multiple readers/writers
    - The normal shell pipe has 1 writer and 1 reader
  - Depends on Unix version, pipes may be **half-duplex**
    - **unidirectional**: with one write end and one read end
  - Or **full-duplex**
    - **bidirectional**: any end for read/write

Unix Pipe: System Calls

Header File	Syntax
`#include <unistd.h>`	`int pipe( int fd[] )`

- Returns:
  - 0 to indicate success; !0 for errors
  - An array of file descriptors is returned:
    - fd[0] == reading end
    - fd[1] == writing end

Unix Pipes: Example Code

```
#define READ_END 0
#define WRITE_END 1

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

   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]);
   }
}
```

Signal (Unix Specific)

- Form of IPC. **Asynchronous notification** regarding an event. Sent to process / thread.
- Recipient of signal must handle signal by: Default set of handlers or user supplied handler.
- **Common UNIX signals**: Kill, Stop, Continue, Arithm. error etc.

Example: Custom Signal Handler

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

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

int main(){
    int *ip = NULL;

    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");

    *ip = 123;

    return 0;
}
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e. the memory segmentation fault signal.

Register our own code to replace the default handler.

This statement will cause a segmentation fault.

6. Synchronization

Problems with Concurrent Execution: When process execute in interleaving fashion AND share a modifiable resource, can cause **synchronization problems**.

- Single sequential process execution deterministic, concurrent processes execution may be non-deterministic.
- **Race Condition**: When final result depends on who runs precisely when.

Critical Regions / Sections

- **Critical Section**: Part of the program where shared memory is accessed (with race condition).
- Incorrect execution is due to **unsynchronized access to shared modifiable resource**.
- If no two processes in their CS at same time, we can avoid races.
- **4 Conditions of good Critical Section / Region (CS) Implementation:**
  1. **Mutual Exclusion**: No two processes simultaneously inside their CS.
  2. **Progress**: If no process in CS, one waiting process granted access.
  3. **Bounded Wait**: No process waits forever to enter its CS.
  4. **Independence**: No process outside CS may block any process.
- **Symptoms of Incorrect Synchronization:**
  - **Deadlock**: All processes blocked, no progress.
  - **Livelock**: Process keep changing state to avoid deadlock, make no progress (dl avoidance mechanism), typically process not blocked.
  - **Starvation**: Some processes blocked forever.

Critical Section Implementations

- **Assembly Level Implementations:** Mechanism provided by processor.
- **High Level Language Implementations:** Utilizes only normal programming constructs. (E.g. CS lock using normal variables)
- **High Level Abstraction:** Abstracted mechanisms that provide additional useful features (E.g. abstract data types, provided as library calls, and involve system calls).

6.1 Assembly Level Implementation

Test and Set (TSL) Instruction

Common machine instruction to aid synchronization: TSL RX,LOCK. (Test and Set Lock). `TestAndSet Register , MemoryLocation`

- **Behavior**: Load current content at **MemoryLocation** into **Register**, and stores a 1 into **MemoryLocation**
- **Atomic**: Performed as a single machine operation, indivisible.
- Assume equivalent high level lanugage ver of **TSL**:

Using Test and Set

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

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

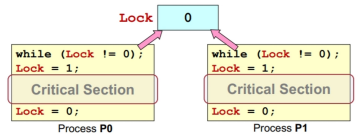
```
void ExitCS( int* Lock )
{
   *Lock = 0;
}
```

- **How Implementation works**: Before entering CS, a process calls enterCS, busy wait until lock is free; then it acquires the lock and returns. After leaving CS process calls ExitCS, which stores a 0 in lock.
- **Inefficient**: Employs busy waiting, wasteful use of processing power.
- **Variants of instruction**: `CompareandExchange` , `AtomicSwap` , `Load Link` .

6.2 High Level Language Implementation

Bad: Lock Variables

Have a single, shared (lock) variable, initially 0. When entering CS, test lock and sets to 1.Thus, 0 means no process in CS, 1 means some process in CS.

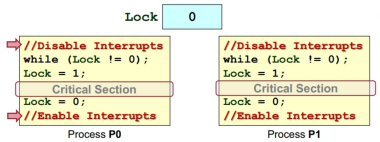


doesn't work properly ☹

Suppose that one process reads the lock and sees that it is 0.
Before it can set the lock to 1, another process is scheduled, runs, and sets lock to 1. When the first process runs again, it will also set lock to 1.
Two processes will be in their critical regions at the same time.

Bad: Lock Variables with Interrupts Disabled

Solve the problem by preventing context switch, by disabling interrupts.

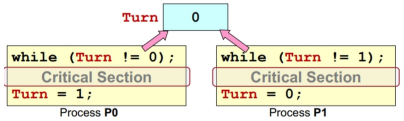


- Buggy critical section may stall the WHOLE system
- Busy waiting
- Requires permission to disable/enable interrupts

Unwise to give user processes power to turn off interrupts.
If system is multiprocessor, disables only one CPU, others will continue to run and access shared memory.

Bad: Strict Alternation

The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

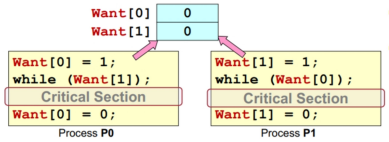


- **Assumption:**
  - P0 and P1 executes the above in loop
  - Take turn to enter **critical section**
- **Problems:**
  - Starvation:
    - E.g. If P0 never enters CS, P1 starve
  - Violate the **independence** property!

Independence property violated: After P0 enters CS, sets turn to 1, and wants to enter CS again, needs to wait for P1 to enter its CS to set turn to 0. If P1 does not enter CS, turn stuck at 1, P0 starves.

Bad: Sub-Peterson Algorithm (without turn)

Leading up to peterson's algorithm. Use global shared array Want, if process wishes to enter CS, set Want[n] to 1.



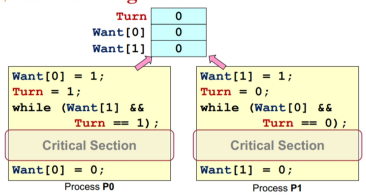
- Solve the independence problem
  - If P0 or P1 is not around, another process can still enter CS
- Problem:
  - Deadlock! Try identify the execution sequence that causes deadlock

If P0 sets Want[0] = 1, preempted, P1 sets Want[1] = 1, both will get stuck and deadlock situation occurs.

Good: Peterson's Algorithm

Assume that writing to Turn is an **atomic operation**.

Peterson's Algorithm



- Busy Waiting:
  - The waiting process repeatedly test the while-loop condition instead of going into blocked state:
- Low level:
  - Higher-level programming construct is desirable
    - simplify mutual exclusion
    - less error prone
- Not general:
  - General synchronization mechanism is desirable
    - Not just mutual exclusion

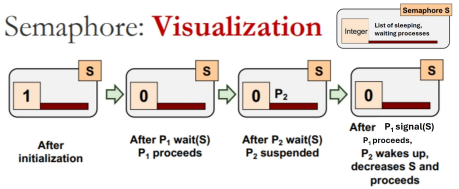
Process can store their process number, or the other process number in turn. Consider both processes trying to enter CS almost simultaneously. Both store in turn variable. Whichever store done last is the one that counts; first one overwritten and lost. After, only one can enter CS, the other will loop.

6.3 High Level Abstraction Implementation

Semaphore

Understanding

- E.W. Dijkstra proposed using semaphore to count the number of wakeups saved for future use. (Processes sleep while waiting for shared resource).
- **Atomic**: Once a semaphore operation started, no other process can access semaphore until operation has completed or blocked.



Semaphore: A generalized synchronization mechanism that specify behavior and not implementation. Provides a way to block a number of processes, which will then be known as **sleeping processes**.

- Semaphore S seen as a "protected integer", with non-negative initial value.
- A general semaphore (aka counting semaphore) can have values $S \geq 0$. A binary semaphore or mutex has values $S = 0$ or 1.

Wait(S)

Takes in a semaphore. If semaphore value is ≤ 0, blocks current process. Decrements the value when it proceeds. Atomic operation. Aka **down()**.

Signal(S)

Takes in a semaphore, wakes up one sleeping process (if any) and increments the semaphore value. This is an atomic operation and never blocks. Aka **up()**.

Semaphore Invariant

- $S_{current} \geq 0$
- $S_{current} = S_{initial} + \#signal(S) - \#wait(S)$
- $\#signal(S)$ is number of $signal()$ executed, $\#wait(S)$ is number of $wait()$ operations completed.

Mutex (Binary Semaphore Usage)

- Set binary semaphore $s = 1$, for any process, do **wait(s)** before entering CS, and **signal(s)** after finishing CS.
- S can only be 0 or 1, deduced by semaphore invariant.
- This usage known as **mutex** (Mutual Exclusion)

Mutex: Correct CS - Informal Proof

- Mutual Exclusion:
  - N_{CS} = Number of process in critical section
    = Process that completed `wait()` but not `signal()`
    = `#Wait( S ) - #Signal( S )`
  - $S_{initial} = 1$
  - $S_{current} = 1 + \#Signal(S) - \#Wait(S)$
  - $S_{current} + N_{CS} = 1$
  - Since $S_{current} \geq 0$ → $N_{CS} \leq 1$
- **Deadlock:**
  - Deadlock means **all** processes stuck at `wait( S )`
    → $S_{current} = 0$ and $N_{CS} = 0$
  - But $S_{current} + N_{CS} = 1$
  - →← (contradiction)
- **Starvation:**
  - Suppose `P1` is blocked at `wait( S )`
  - `P2` is in CS, exits CS with `signal( S )`
    - If no other process sleeping, `P1` wakes up
    - If there are other process, `P1` eventually wakes up (assuming fair scheduling)

- **Possible Deadlock**: Note incorrect usage of semaphore may still result in deadlock, e.g. incorrect interleave usage of semaphore.
- **Conditional Variable**: Similar to semaphore, but allows process to wait for some event to happen. Once event happens, broadcast made to wake up all waiting tasks.

6.4 Classical Synchronization Problems

Producer-Consumer Problem

Processes share a bounded buffer of size K. Producers produce items to insert in buffer, only when the buffer is not full. Consumers remove items from buffer, only when the buffer is not empty. *How to sync the two?*

Busy Waiting Solution (Inefficient)

Producer Consumer: **Busy Waiting**

```
while (TRUE) {
    Produce Item;
    while(!canProduce);
    wait( mutex );
    if (count < K) {
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        canConsume = TRUE;
    } else
        canProduce = FALSE;
    signal( mutex );
}
```
Producer Process

```
while (TRUE) {
    while (!canConsume);
    wait( mutex );
    if (count > 0) {
        item = buffer[out];
        out = (out+1) % K;
        count--;
        canProduce = TRUE;
    } else
        canConsume = FALSE;
    signal( mutex );
    Consume Item;
}
```
Consumer Process

- **canConsume**:
  - Triggers consumer to *try* to get item
- **canProduce**:
  - Triggers producer to *try* to produce item
- `wait(mutex) + signal(mutex)` : Creates a CS
- `in = (in+1) % K` :
  `out = (out+1)% K` : Wraps around, circular array
- Initial Values:
  - `count = in = out = 0`
  - `mutex = S(1)` //semaphore with initial value 1
  - `canProduce = TRUE` and `canConsume = FALSE`;
- Evaluation:
  - The code **correctly solve** the problem
  - However, **busy-waiting** is used

Blocking Solution

Solution uses three semaphores, one called (*notFull*) for counting empty slots, one called (*notEmpty*) for counting full slots, and one called *mutex* to make sure producer and consumer do not access the buffer at the same time.

- *NotFull* is initially equal to the number of slots in the buffer, *NotEmpty* is initially 0, and *mutex* is initially 1.
- When producing item, wait(notFull), which decrements. Then, acquire mutex. After producing, signal (release) mutex, and signal(notEmpty), causing it to increment.

Producer Consumer: **Blocking Version**

```
while (TRUE) {
    Produce Item;

    wait( notFull );
    wait( mutex );
    buffer[in] = item;
    in = (in+1) % K;
    count++;
    signal( mutex );
    signal( notEmpty );
}
```
Producer Process

```
while (TRUE) {

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

    Consume Item;
}
```
Consumer Process

- `wait( notFull )` : Forces producers to go to sleep
- `wait( notEmpty )` : Forces consumers to go to sleep
- `signal( notFull )` : 1 consumer wakes up 1 producer
- `signal( notEmpty )` : 1 producer wakes up 1 consumer
- Initial Values:
  - `count = in = out = 0`
  - `mutex = S(1), notFull = S(K), notEmpty = S(0)`
- Evaluation:
  - This code correctly solve the problem
  - No busy-waiting, "unwanted" producer/consumer will go to sleep on respective semaphores

Readers and Writers Problem

Processes share a data structure D, where readers can access and read information from D together, while writers must have exclusive access to D to write information. *How to sync the two?*

Readers Writers: **Simple Version**

```
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`

In this solution, the first reader to get access to the database does a wait (down) on semaphore **roomEmpty**.

Subsequent readers merely increment counter, **nReaders**.

As readers leave, they decrement **nReaders**, and the last to leave does a signal (up) on **roomEmpty**, allowing a blocked writer, if there is one, to get in.

- **Problem**: As long as at least one reader active, subsequent readers admitted. Writer kept suspended until no reader is present. Writer may never get in.
- **Rectification**: Program could be written slightly differently: when reader arrives and a writer is waiting, the reader suspended behind writer instead of being admitted immediately. Writer waits for readers active to finish, but not readers after. Disadvantage is less concurrency and thus lower performance.

Dining Philosophers

Five philosophers are seated around a table, and there are five single chopsticks placed between each pair of philosophers. When any philosopher wants to eat, he/she will have to acquire both chopsticks from his/her left and right.
How can we have a deadlock-free and starvation-free way to allow the philosophers to eat freely?

Obvious Wrong Solutions

- **Wait till fork is available**: Obvious solution is wrong. If all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.
- **After taking left fork, check right fork. If not available, put down left one, wait, and repeats**: Also fails, for different reason, if all philosophers start algorithm simultaneously, will cause livelock, fail to make progress, cause starvation.
- **Use single mutex**: Before acquiring forks, wait(mutex), after putting down, signal(mutex). While no deadlock, no starvation, It has perfomance bug, only one can eat at an instant. With five forks, at least two philosophers can eat at same time.

Tanenbaum Solution

The solution presented is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers.

- It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire cultery forks).
- A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.
- The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy.

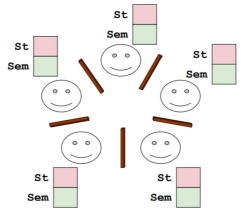
Dining Philosopher: **Tanenbaum Solution**

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

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

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

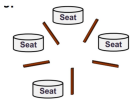
```
void putChpStcks( i )
{
    wait( mutex );

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

    signal( mutex );
}
```



Limited Eater Solution

Dining Philosopher: **Limited Eater**

- If **at most 4 philosophers are allowed to sit at the table (leaving one empty seat)**
  ➔ **Deadlock is impossible!**



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

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

6.5 Synchronization Implementations

POSIX Implementations in Unix:

POSIX Semaphore

- Popular implementation of semaphore under Unix
- Header File:
  - `#include <semaphore.h>`
- Compilation Flag:
  - `gcc something.c -lrt`
  - Stand for "real time library"
- Basic Usage:
  - Initialize a semaphore
  - Perform `wait()` or `signal()` on semaphore

pthread Mutex and Conditional Variables

- Synchronization 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()`

- Programming languages with thread support will have some forms of synchronization mechanism
- **Examples:**
  - Java: all object has built-in lock (mutex) (monitor), synchronized method access, etc
  - Python: supports mutex, semaphore, conditional variable, etc
  - C++: Added built-in thread in C++11; Support mutex, conditional variable

7.