Processes

CSD2180 Operating Systems
BSc in Computer Science (IMGD / RTIS)
Singapore Institute of Technology / DigiPen Institute of Technology
September 2021

SINGAPORE INSTITUTE OF TECHNOLOGY

Attendance Taking

https://forms.office.com/r/rC3TKbsgpR

- Log in to your SIT account to submit the form
- You can only submit the form once
- The codeword is **tungsten**

Week 2 Quiz Details

Open book

No need to use the Respondus Lockdown Browser

Process Concept

A process is a program in execution

The Process

An operating system executes a variety of programs that run as a process

Program is a **passive** entity stored on disk (**executable file**); process is **active**

Program becomes process when an executable file is loaded into memory

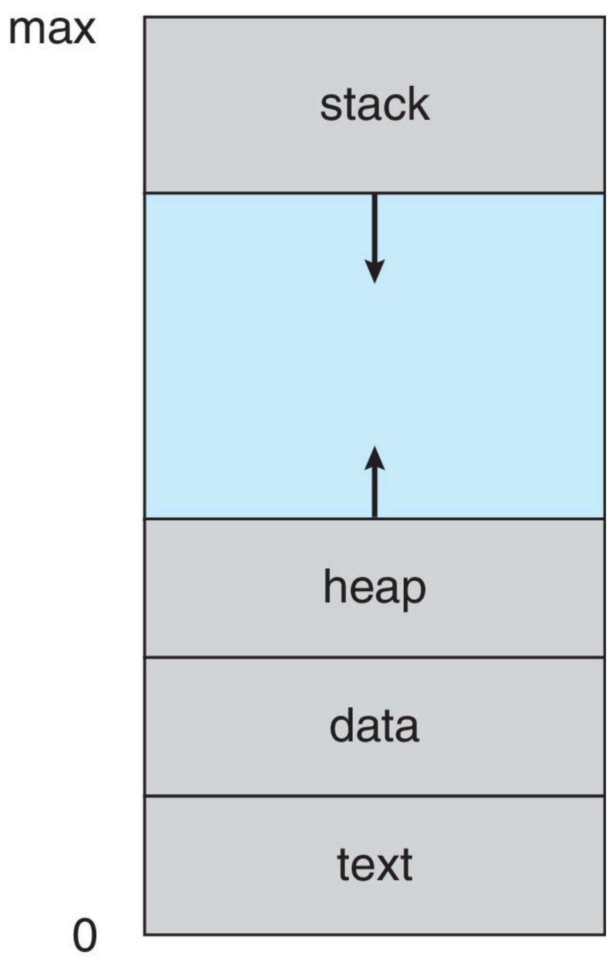
Process

- A program in execution
- Process execution must progress in sequential fashion
- No parallel execution of instructions of a single process

One program can be several processes

- Consider multiple users executing the same program

Process in Memory



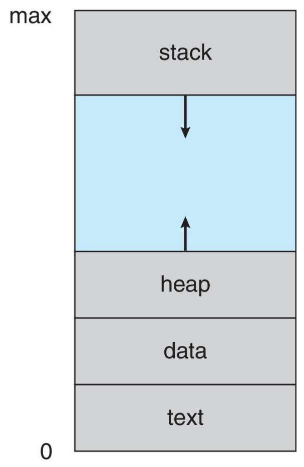
Text section: Comprises the compiled program code (from non-volatile storage) when launched

Stack: Contains temporary data such as function parameters, return addresses, local variables

Data section: Contains global and static variables, allocated and initialized prior to executing main

Heap: Contains memory dynamically allocated during run time and is managed via calls such as malloc, free, etc.

Process in Memory



Stack and heap grow toward one another; the operating system must ensure they do not overlap one another

Should they ever meet

- Stack overflow error will occur
- A call to new or malloc will fail due to insufficient memory available

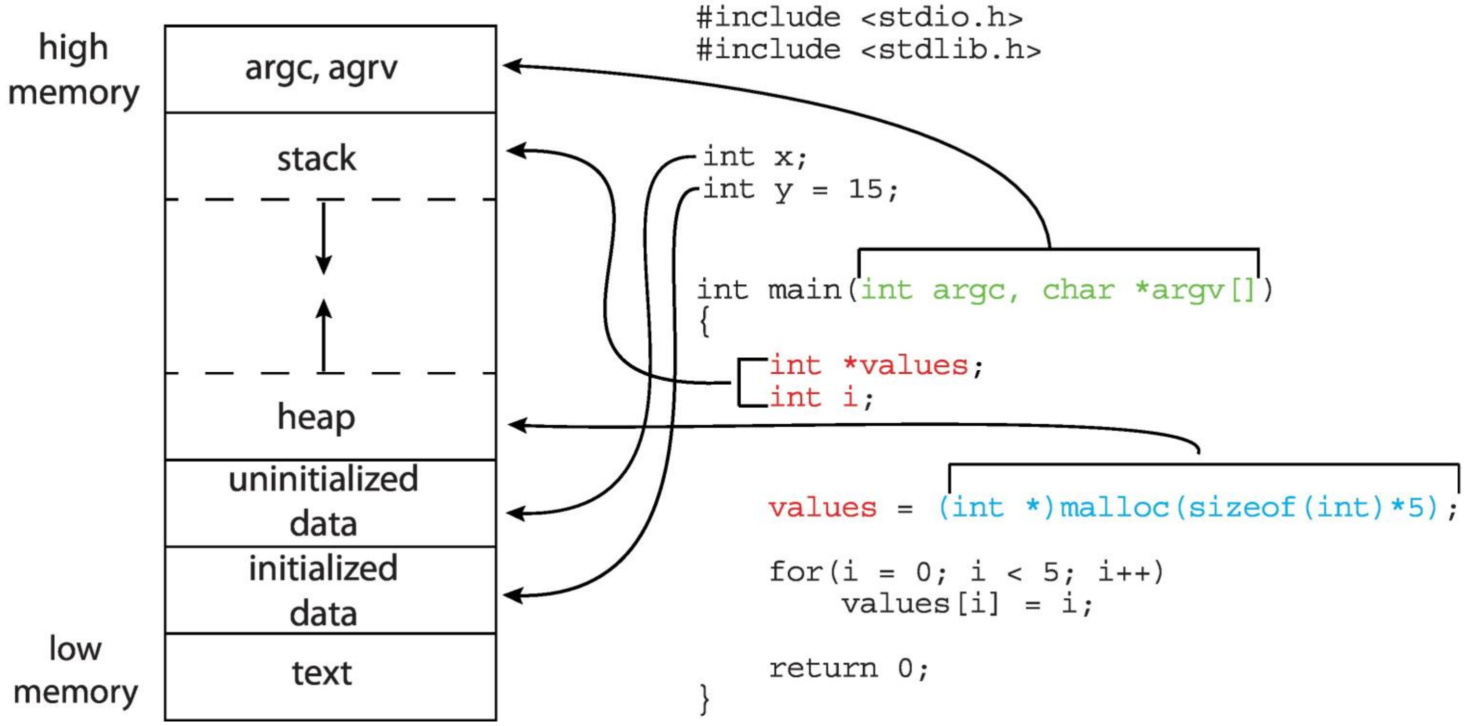
Some notes about process in memory

- Sizes of text and data sections do not change during program run time
- Stack and heap sections can shrink and grow dynamically during execution
- Each time a function is called, an **activation record** containing function parameters, local variables, and return address is pushed onto the stack
- When control is returned from the function, the activation record is popped from the stack
- Heap will grow as memory is dynamically allocated, and will shrink when memory is returned to the system

Memory Layout of a C Program

Similar to the previous diagram, except

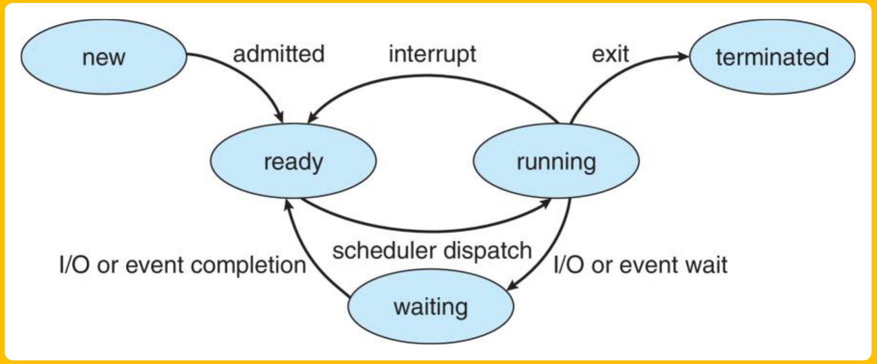
- The data section is divided into distinct sections for initialized and uninitialized data
- A separate section is provided for the *argc* and *argv* parameters passed to the main() function



Process State

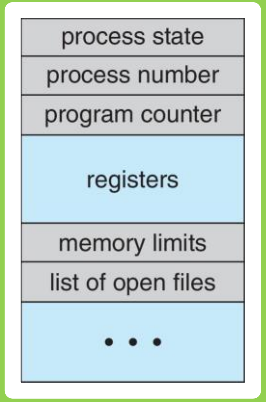
A process changes state as it executes

- **New:** The process is being created
- **Ready:** The process is waiting to be assigned to a processor
- **Running:** Instructions are being executed
- **Waiting:** The process is waiting for some event to occur
- **Terminated:** The process has finished execution



Protip: Many processes may be ready and waiting, but only one process can be running on any CPU core at any instant

Process Control Block



PCB serves as the repository for all data needed to start, or restart, a process, along with some accounting data

Information associated with each process

- **Process state:** Running, waiting, etc.
- **Process number:** Process ID and parent process ID
- **Program counter:** Location of instruction to next execute
- **CPU registers:** Contents of all process centric registers
- **CPU scheduling information:** Priorities, scheduling queue pointers
- **Memory management information:** Memory allocated to the process
- **Accounting information:** CPU used, clock time elapsed since start, time limits
- **I/O status information:** I/O devices allocated to process, list of open files

Process Scheduling

When can the operating system switch the CPU from one process to another? 🔥

Which process should it switch to? 🔥

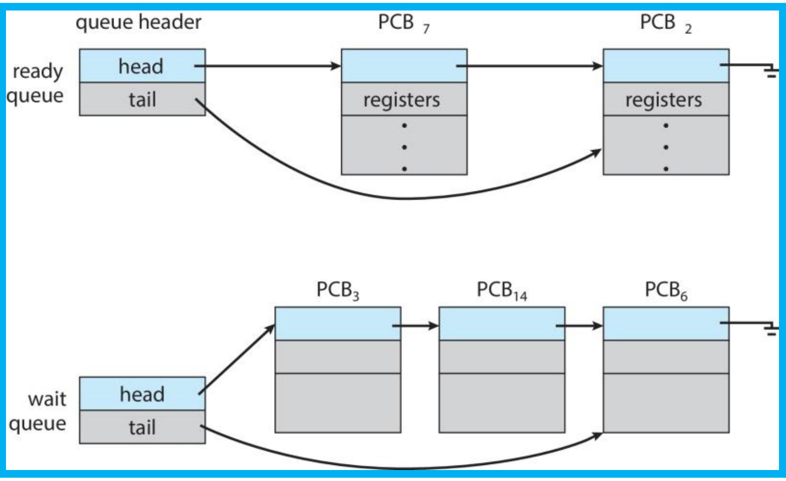
Process Scheduling

Process scheduler selects among available processes for next execution on CPU core

Maintains scheduling queues of processes

- **Ready queue:** Set of all processes residing in main memory, ready and waiting to execute
- **Wait queues:** Set of processes waiting for an event (i.e., I/O)

Processes migrate among the various queues



Goal: Maximize CPU use, quickly switch (ready) processes onto CPU core

Process Scheduling

Goal: Maximize CPU use, quickly switch (ready) processes onto CPU core

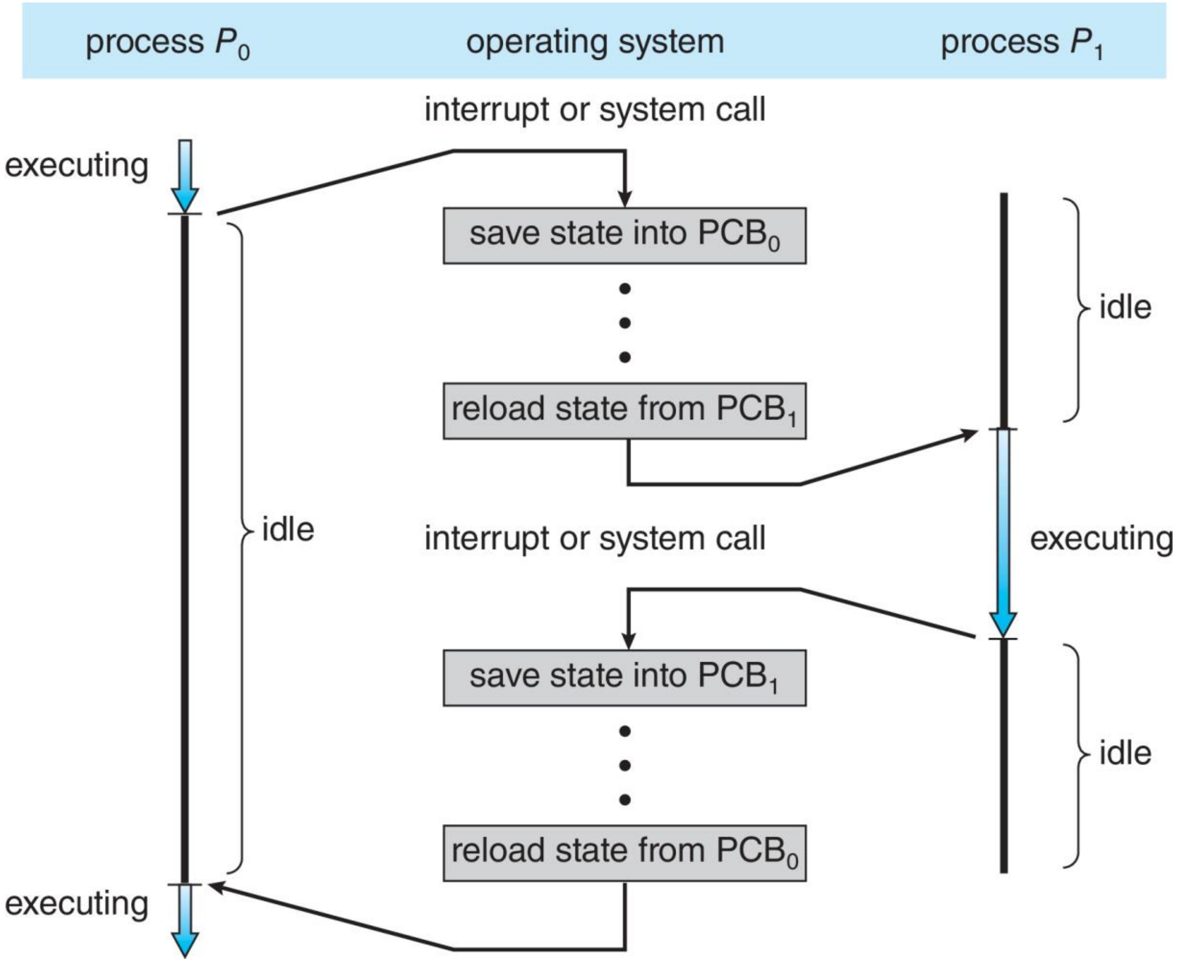
Processes can be described as either

- **I/O bound:** Process spends more time doing I/O than computations; many short CPU bursts
- **CPU bound:** Process spends more time doing computations; few very long CPU bursts

Constraints and considerations

- There is a need to deliver "acceptable" response times for all programs, particularly interactive ones
- Process scheduler must implement suitable policies for swapping processes in and out of the CPU
- Every time the operating system steps in to swap processes, it takes up CPU time which is "lost" for doing any productive work

Context Switch



When CPU switches to another process, the system must **save the context** of the old process and **load the saved context** for the new process via a context switch

Context of a process represented in the PCB

Context switching time is **pure overhead**; the system does no useful work while switching

The more complex the operating system and the PCB, the longer the context switch takes

Operations on Processes

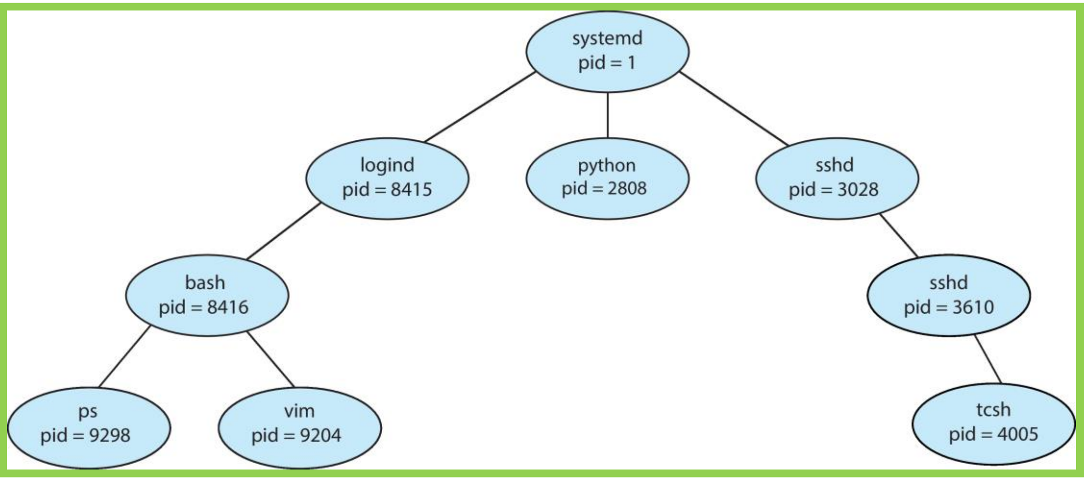
Operating system must provide mechanisms for process creation and process termination

Process Creation

A process may create new processes over its course of execution

The creating process is called the **parent** process; the new processes are called the **child** of that process

Each of these new processes may in turn create other processes, forming a tree of processes



Processes are generally identified and managed via a *process identifier* (PID)

Process Creation

Resource sharing options

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Execution options

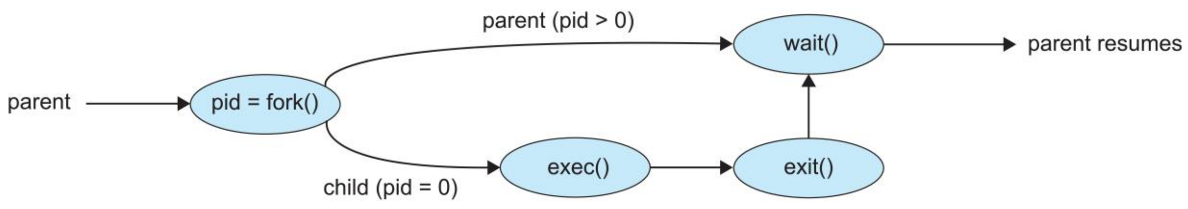
- Parent and children execute concurrently
- Parent waits until children terminate

Address space options

- Child duplicate of parent
- Child has a program loaded into it

Creating a Process in Unix

- **`fork()`** system call creates new process
- **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
- Parent process calls **`wait()`** waiting for the child to terminate



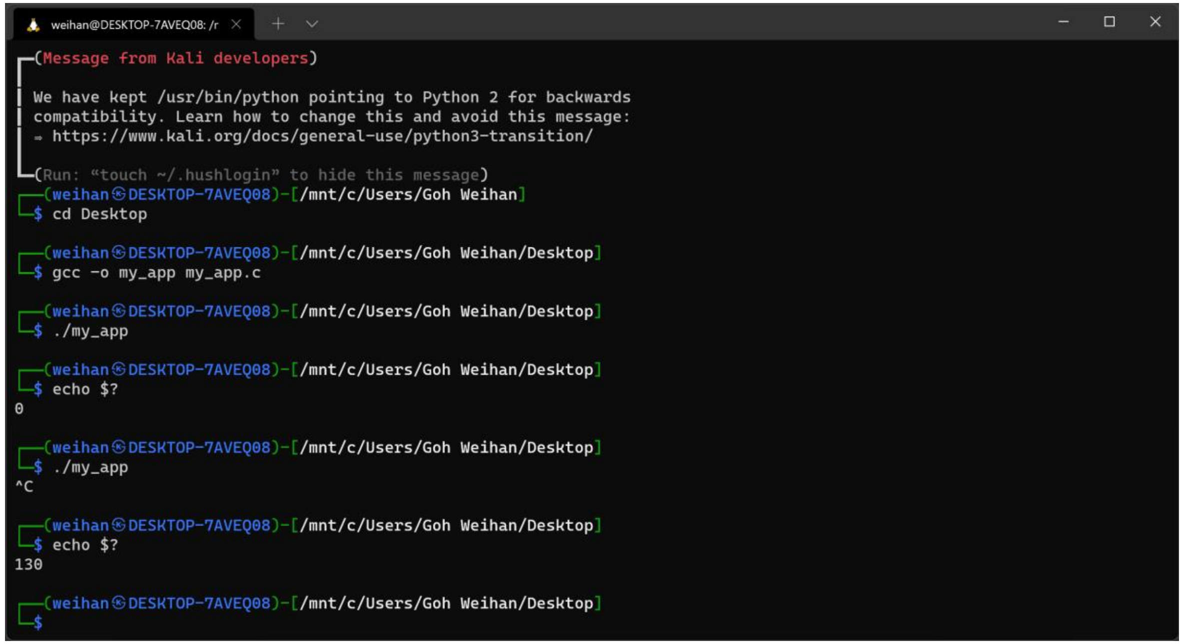

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    /* return 0 to child process, non-zero child PID to the parent */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    } else if (pid == 0) { /* child process */
        /* execlp() is a version of the exec() syscall */
        execlp("/bin/ls", "ls", NULL);
    } else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete"); /* any last words? */
    }

    return 0;
}
```

Process Termination

A process terminates when it finishes executing its **final statement** and asks the operating system to delete it by using the **`exit()`** system call

The process may return a **status value** to its waiting parent process (via **`wait()`**)

Process resources, including physical and virtual memory, open files, and I/O buffers, are deallocated and reclaimed by the operating system



Process Termination

Parent may terminate the execution of child processes for various reasons

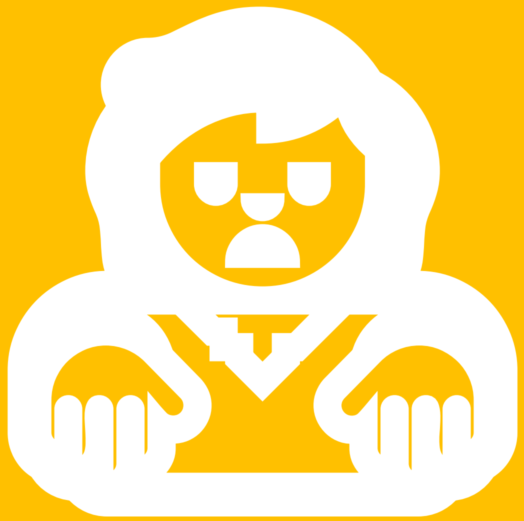
- Child has **exceeded allocated resources**
- Task assigned to child is **no longer required**
- The parent is exiting, and the operating systems **does not allow a child to exist** if its parent terminates

Some operating systems do not allow child to exists if its parent has terminated

- If a process terminates, then all its children must also be terminated
- **Cascading termination:** All child processes, grandchild processes, etc., are terminated
- Such termination is initiated by the operating system

Process Termination

A zombie process is one that has terminated, but whose parent has yet to invoke `wait()`



Orphaned processes are child processes of a parent that terminated without invoking `wait()`

What happens to orphaned processes?

Interprocess Communication

Interprocess Communication

Processes within a system may be independent or cooperating

- **Independent processes** cannot affect or be affected by the execution of another process
- **Cooperating processes** can affect or be affected by other processes, including sharing data

Two models for interprocess communication

- **Shared memory**
- **Message passing**

Reasons for cooperating processes

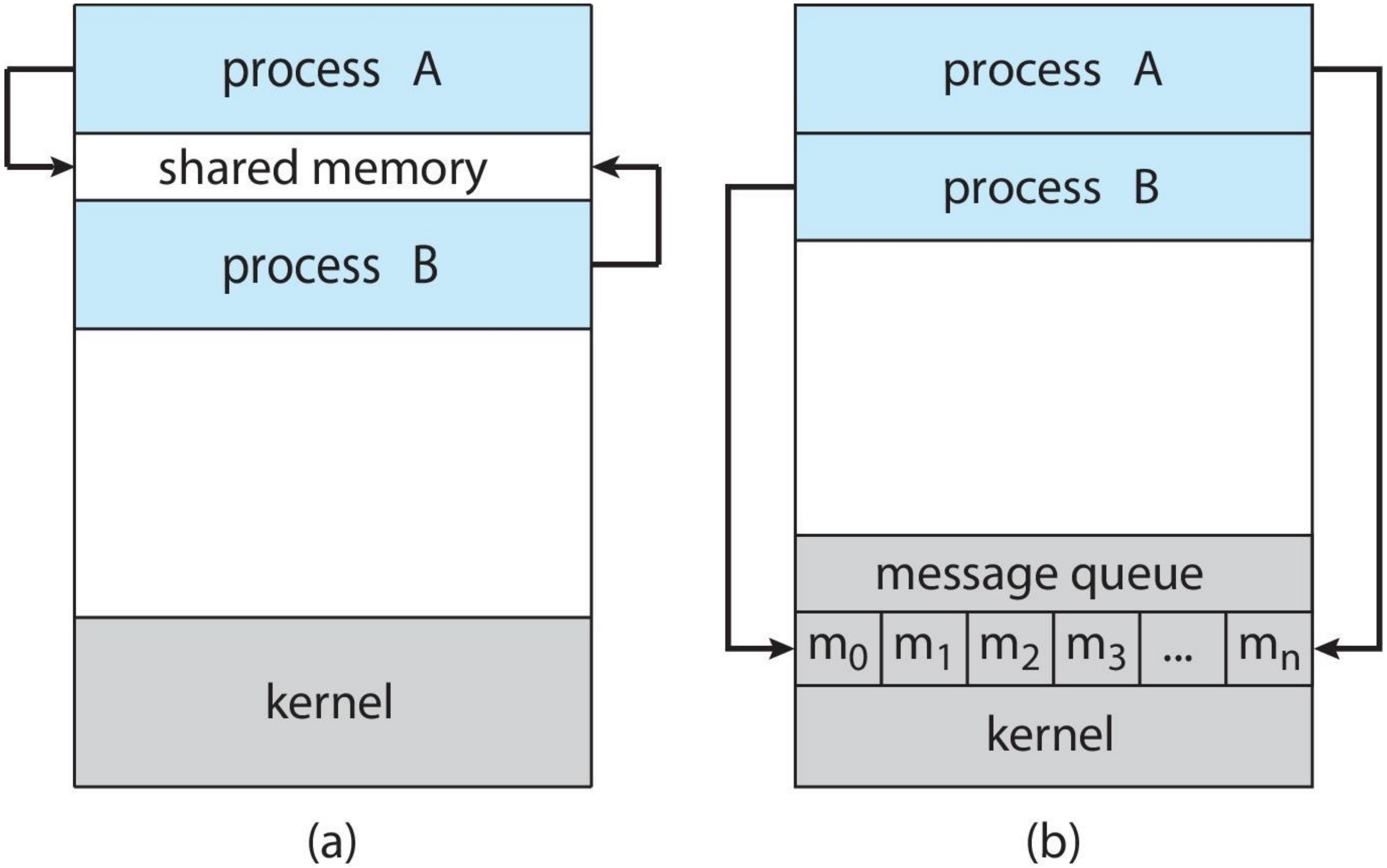
- **Information sharing**
- **Computation speedup**
- **Modularity**
- **Convenience**

Cooperating processes need interprocess communication (IPC)

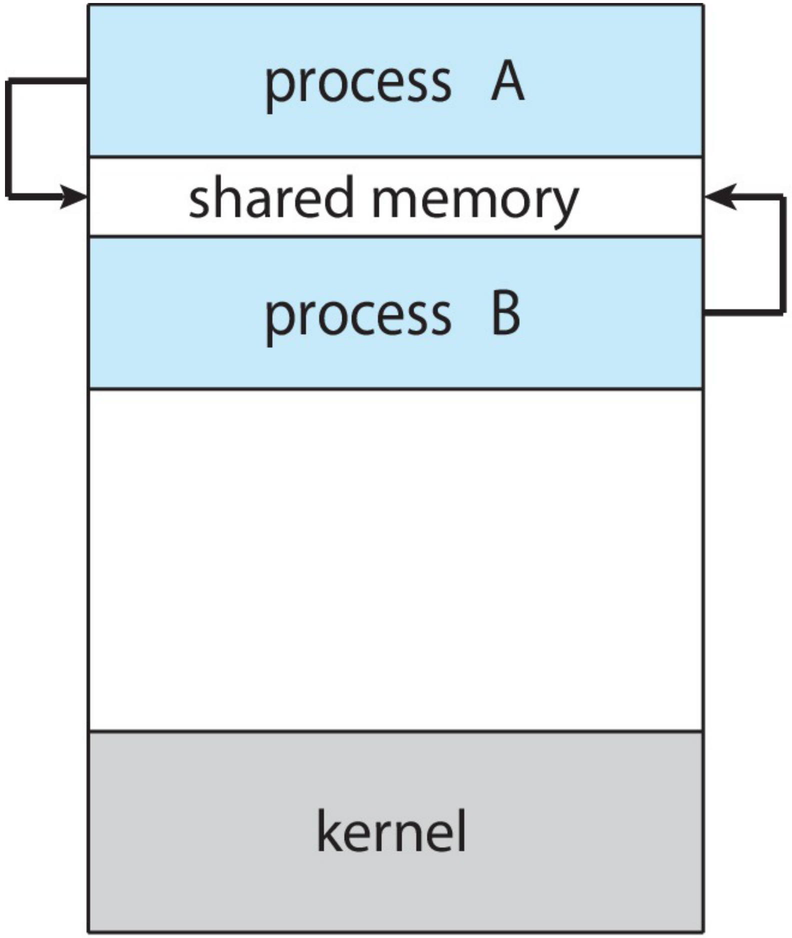
Interprocess Communication Models

Two models for interprocess communication

- Shared memory (a), where a region of memory is shared by the cooperating processes
- Message passing (b), where messages exchanges happen between the cooperating processes



Shared Memory



An area of memory shared among processes that wish to communicate

Communication is controlled by the processes, not the operating system

What can possibly go wrong? 🤔

What Can Possibly Go Wrong with Shared Memory?

- A process tries to write to an already full shared memory buffer
- A process tries to read from an empty shared memory buffer
- Multiple processes try to access the shared memory buffer at the same time

How do we solve this? 🤔

Let's take a step back... 🧋

Producer-Consumer Problem

A model for cooperating processes

Producer process produces information

Consumer process consumes information

Two variations of the producer-consumer problem

- **Unbounded-buffer**, i.e., no limit on the buffer size
  - Producer never waits
  - Consumer waits if there is nothing in buffer to consume
- **Bounded-buffer**, i.e., fixed buffer size
  - Producer must wait if buffer is full
  - Consumer waits if there is nothing in buffer to consume

Message Passing

Processes communicate with each other by passing messages to each other

IPC facility provides two operations

- `send(message)`
- `receive(message)`

If two processes wish to communicate, they need to

- Establish a **communication link** between them
- Exchange messages via send / receive

Implementation considerations

- Direct or indirect communication?
- Synchronous or asynchronous communication?
- Automatic or explicit buffering?

Direct and Indirect Communication in Message Passing

Direct communication

- Processes must name each other explicitly
  - **`send(P, message)`:** Send message to process P
  - **`receive(Q, message)`:** Receive message from process Q
- Properties of direct communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect communication

- Messages are directed and received from *mailboxes* (also called *ports*)
  - Each mailbox has a **unique id**
  - Processes can communicate only if they share a mailbox
- Properties of indirect communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

A Note on Indirect Communication in Message Passing

Operations

- Create a new mailbox (port)
- Send and receive messages through mailbox
- Delete a mailbox

How to communicate?

- `send(Z, message)`: Send message to mailbox Z
- `receive(Z, message)`: Receive message from mailbox Z

Mailbox sharing

- Problem
  - P, Q, and R share mailbox A
  - P sends; Q and R receive
  - Who gets the message?
- Possible solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver; sender is then notified who the receiver was

Synchronization

Message passing may be either blocking or non-blocking

Different combinations possible

If both send and receive are **blocking**, we have a **rendezvous**[*] between sender and receiver

[*] Rendezvous: In interprocess communication, when blocking mode is used, the meeting point at which a send is picked up by a receive

Blocking is considered synchronous

- **Blocking send:** The sender is blocked until the message is received
- **Blocking receive:** The receiver is blocked until a message is available

Non-blocking is considered asynchronous

- **Non-blocking send:** the sender sends the message and continue
- **Non-blocking receive:** the receiver receives either a valid message, or null message

Buffering

Messages exchanged by communicating processes reside in a temporary queue

Such queue can be implemented in three ways

- **Zero capacity:** Queue has a maximum length of zero; sender must wait for receiver
- **Bounded capacity:** Queue has finite length *n*; sender must wait if link is full
- **Unbounded capacity:** Queue length is (potentially) infinite; sender never waits

SINGAPORE INSTITUTE OF TECHNOLOGY

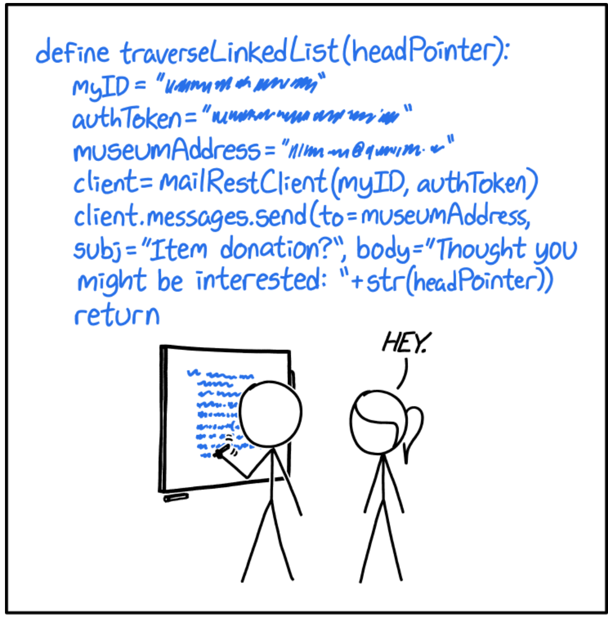
Questions?

Thank You!

Weihan.Goh {at} Singaporetech.edu.sg

https://www.singaporetech.edu.sg/directory/faculty/weihan-goh

https://sg.linkedin.com/in/weihan-goh



CODING INTERVIEW TIP: INTERVIEWERS GET REALLY MAD WHEN YOU TRY TO DONATE THEIR LINKED LISTS TO A TECHNOLOGY MUSEUM.