Can pass parameters to OS by passing parameters to registers OR passing a block to a register OR pushing/popping onto stack

Policy: what will be done? (flexible definition)   **Mechanism**: how to do it? (rarely change)

Process: active entity loaded into memory for execution and is allocated a processor CPU and other resources

- Memory layout: **text**, **data**, **heap**, **stack**
- States: **new**, **running**, **waiting**, **ready**, **terminated** (can move from waiting queue to ready queue)
  - Blocked process is put into wait queue
  - Interrupts can remove process and put into ready queue
- Information stored in PCB (maintains process state, program counter, CPU registers, etc)
  - Used when **context switching** from the current process to a new process (process state is pushed on PCB)
- Can be **I/O bound** or **CPU bound**
- **Swapping** can be done to reduce **degree of multiprogramming**
- Parent-Child relationships:
  - OS allocates new resources to child OR child takes a subset of parent resources
  - Parent can concurrently execute with its child OR wait until some or all of its children have terminated
  - Child process can share a copy of parent address space OR load a new program (`exec()`)
  - **Zombie**: process has terminated but parent hasn't called `wait()`   **Orphan**: process with no parents
  - When parent terminates, can SOMETIMES result in **cascading termination**

IPC: provides **information sharing**, **computation speedup**, and **modularity** between **cooperating processes**

- **Shared memory** (read/write to shared area) or **message passing** (send/receive messages from **communication link**)
- **Producer-Consumer** can use **unbounded buffer** or **bounded buffer** (processes maintain `in` and `out` variables)
- **Direction Communication** requires explicit naming, **Indirect Communication** has messages sent to a **mailbox**
  - **Synchronous** (blocks until message is received or available) or **Asynchronous** (receiver can get message or `NULL`)
  - **Zero**, **Bounded**, or **Unbounded Capacity** on the message queue

---

Threads: basic unit of CPU utilization that shares data with other threads belonging to the same process

- Shares code section, data section, and OS resources. PCB maintains PC pointers for each thread (**Multithreading**)
- Provides **Responsiveness**, **Resource Sharing**, **Economy**, and **Scalability** (for multicore environment)
- **Concurrency** means thread execution is interleaved   **Parallelism** means threads are executed in parallel
  - Types of parallelism: **Data Parallelism** and **Task Parallelism**
- Multithreading models used for when user thread make call to kernel thread
- Thread libraries implemented in user space (uses local function calls) OR in kernel space (uses system calls)
- **Asynchrounous Threading** usually involves little data sharing. **Synchronous Threading** usually has a lot of data sharing

Implicit Threading: abstracts the complexities of creating and mapping tasks to separate threads for execution

- **Thread Pools**: create finite number of threads that wait for a request (don't have to waste time creating new threads)
- **OpenMP**: programmer identifies regions that can be run in parallel using compiler directives
- **Grand Central Dispatch**: tasks are placed in **serial queue** or **concurrent queue** and are eventually assigned a thread

Threading Issues: `fork()/exec()` semantics, **signal handling**, **thread cancellation**, **TLS**, **scheduler activations**

- Receiving signals **Synchronous** (from same process) or **Asynchronously** (from another process)
- Semantics of sending signals to multithreaded programs (all threads or just a few of them)
- **Asynchronous** (immediately) or **Deferred** cancellation (target thread checks itself)
- Scheduler activations use **LWP**s to schedule user thread to run attached to a kthread. Kernel inform apps through **upcalls**

Systems consist of manages **safety** (objects across multiple activities) and **liveness** (span across multiple objects)

Critical-Section Problem: must satisfy **Mutual Exclusion**, **Progress**, and **Bounded Waiting**

Single cores can just disable interrupts during critical section. For multiprocessors, use **preemptive** or **nonpreemptive kernels**

Peterson's Solution: manages 2 shared variables: `int turn` and `bool flag[2]` (process is ready to enter CS). Only for 2 processes

```
flag[i] = true;
turn = j;
while (flag[j] && turn == j);    // busy wait

/* critical section */

flag[i] = false;

/* remainder section */
```

Bakery's Solution: before CS, each P_i receives a number. P with smallest number enters CS. If equal, then $i < j \implies P_i$ before P_j

Atomic instructions either: test word and set value OR swap contents of 2 words

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

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

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

Mutex: process calls `acquire()` and `releases()` before entering/leaving critical section. Results in **busy waits**.

Semaphores: integer variable accessed through atomic operations `wait()` and `signal()` (**counting** vs **binary**)

- **Note**: moves lock management to critical section (potential issue w/ busy waits), whereas mutex had them before and after CS
- Can force P_2 to execute S_2 only AFTER P_1 executes S_1 (`signal()` called after S_1 and `wait()` called before S_2)
- Instead of busy waits, can **suspend** process: move it to waiting queue. Can lead to **deadlock**, **starvation**, **priority inversion**

Monitors: abstract functions for process synchronization (only 1 P can enter monitor at a time)

- Maintains condition variables: P_i that invokes `x.wait()` is suspended until `x.signal()` invoked by another P_j
- **Note**: `x.signal()` only affects if another P is waiting. For semaphore, `signal()` always affected semaphore state
- 2 options of **signal and wait** OR **signal and continue**
- **Conditional wait** can be used to select next P to resume in monitor where P with lowest number (highest priority)