Terms

- **OS** provides the environment within which programs are executed
  - Provides *services* for programs
  - Provides *interfaces* for users
  - Collection of *components* and their interconnections
- **Parallelism** > 1 task being performed simultaneously (one process may be waiting on another)
- **Concurrency** multiple tasks making progress at one time (nobody is waiting)
- **Throughput** number of processes completed in a unit of time
- **Turnaround** from time of submission to completion, INCLUDING wait time
- **Wait Time** time spent in waiting/ready queue, NOT including I/O queue / time
- **Response Time** time from submission to time when 'first' usable data/output produced
- **TLB** Translation Lookaside Buffer
- **MMU** Memory Management Unit
- **RMS** Rate Monotonic Scheduling
- **EDF** Earliest Deadline First
- **PTBR** Page Table Base Register (pointer to page table in memory)

Parameter Passing

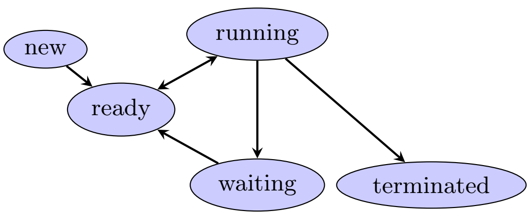
1. Pass paramater via registers
2. Save parameters in block/table (memory), pass via registers the address of the block
3. Placed onto the stack

System Calls

Several basic 'types'

1. Process Control end, abort, load create process ...
2. File Management create, delete, open files, ...
3. Device Management request, write to device, ...
4. Info Management get process attr, set sys data
5. Communication create, delete, send, receive messages
6. Protection prevent read, allow modification by owner, ...

Process



New being created
Running being executed
Waiting for some event
Ready Waiting to be assigned to a processor
Terminated Finished execution

State
Program Counter
CPU Registers
CPU Scheduling Info
Mem Mgmt Info
Accounting Info
I/O Status

Two processes are *independent* if the write set of each is disjoint from both the read and write sets of the other.

Threads

Many-to-one

All user threads map to a single kernel thread
If a user thread makes a block system call, the entire process (made up of multiple user threads) will block Because only 1 thread can access the kernel at any one time, multiple threads are unable to run concurrently on a multicore computer

One-to-one

Each user thread is mapped to a unique kernel thread
The creation of a user thread requires (considerable) overhead to create a kernel thread. If a user thread is idle (perhaps waiting on another thread to finish), then the kernel thread with which the user thread is associated is needlessly consuming kernel space resources When one thread is blocked (user or kernel thread), the other threads can continue. Concurrency is enabled

Many-to-many

Many user threads are mapped to a $\leq$ # kernel threads At the user level, multiple threads are created, and it is up to the OS to schedule/orchestrate/map each of them to a kernel thread

With m threads with n instructions each:

$$\frac{(mn)!}{(n!)^m} \text{ possible histories}$$

Critical Section Problem

Mutual Exclusion If a process is executing its critical section, no others can be executing theirs

Progress If no process is executing its critical section, AND some process wants to enter its, only those processes NOT executing can decide who enters

Bounded Waiting There must be a limit on the number of times another process is allowed to enter its critical section after a process has made a request to enter its critical section (i.e., no starvation).

Semaphores

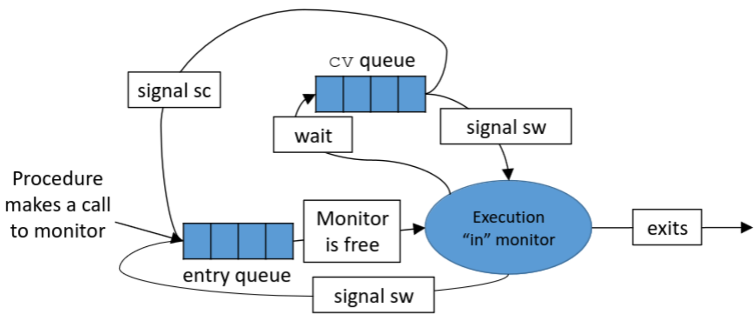
- Must be careful NOT to omit an inc/dec in code.
- Global, thus must know how entire program works to use them
- Can't infer which waiting process will run next

Monitors

SC The signaler continues, and the signaled executes at some later time

SW The signaler waits until some later time and the signaled executes immediately

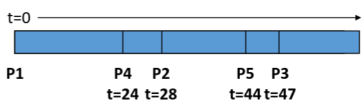
A monitor follows one, but not both.



Scheduling

FCFS First Come First Serve
SJF Shortest Job First
Priority Highest priority first

PID	ms	Priority
P1	24	1
P2	16	3
P3	2	5
P4	4	2
P5	3	4



Priority

May encounter issues like starvation / aging. Hence...

Round Robin

- Circular queue
- Time quantum
- Each process has a given burst time

Real-time scheduling

Periodic occuring at a constant interval / period (p)
Process time time needed to burst (t)
Deadline time the process must be completed by (d)

If a requesting process does not 'satisfy' $0 \leq t \leq d \leq p$, the scheduler should reject the process.

Suppose two processes:

Process 1 p:50ms, t:20ms, d:1/period
Process 2 p:100ms, t:35ms, d:1/period

Can the CPU process both?

Utilization (t/p)

$$\begin{aligned} P1: 20/50 &= 0.4 \\ P2: 35/100 &= 0.35 \\ & 0.75 \end{aligned}$$

Assuming no other process runs, we should be able to service both.

RMS Upon entering, a process is assigned a priority inversely proportional to its period. (i.e., the shorter the period, the higher the priority). P2 would be broken up into two chunks (30 / 5) at t=50.
EDF Upon entering, a process is assigned a priority inversely proportional to its deadline. (i.e., the sooner the deadline, the higher the priority).

By design, real-time CPU scheduling does not permit a process that has already met its period deadline to start a second time in the same period.

Pipeline Stages

- (F)etch
- (D)ecode
- (E)xecute
- (W)rite (B)ack

Hard Drive
Memory
L2 Cache
L1 Cache
Registers
ALU

Memory Allocation

Best Fit Fit into a hole such that resulting left-over hole is size minimized (ideally 0)
First Fit Fit into first hole (most often reading from bottom addresses to higher) that can accomodate the process
Worst Fit Fit into largest hole, resulting in left-over hole whose size is maximized

Segmentation

Logical Address −> <segment num, offset>

$$\text{Logical Address} -> \begin{cases} \text{Segment Number, offset} \\ S, D \end{cases}$$

S is the index into the table which identifies a row containing limit and base value. In the example table, <3, 200>, <3, 60000> would result in address errors

Segment Table

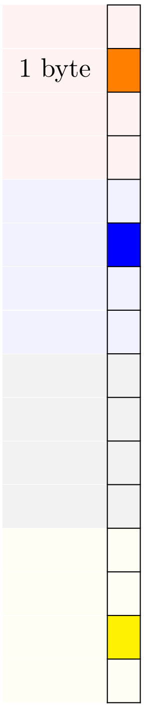
limit	base
1400	2500
2600	3500
17000	1800

Fragmentation

External occurs when unused space is non-contiguous
Internal refers to the unused space within a frame / page

Paging

A page size is a power of 2, usually between 512 bytes and 1 GB.



The $m - n$ highest order bits are needed to address into the page table.

$$\begin{aligned} \text{Logical Address Space} &= 2^m = 2^4 = 16 \\ \text{Page Size} &= 2^n = 2^2 = 4 \text{ bytes} \end{aligned}$$

	p	d
	00	01
	01	01
	11	10

Thus, the logical address gives us some (p, d). We use p to index into the page table to find the correct frame (f), with the offset of d (which is from the n lower bits of the address). Thus our physical address is (f, d).

A single datum in logical address space still requires reserving the entire frame for that datum. Internal fragmentation refers to the unused space within a frame/page.

Downsides

- Every mapping MUST go through page table
- A page table can be implemented via registers, but this limits their size
- One solution is putting it in main memory, and having the PTBR point to where it is

Inverted Page Table

A SINGLE page table regardless of how many processes. CPU generates a pid, page number, and offset.

Search the entire page table for <pid, p>. The *index* of that entry is the frame number.

Inverted Page Table

pid	p

Miscellaneous

If segmentation is used, and the total free (unallocated) main memory is 128MB, then the OS can place into main memory a segment of size 64MB.

False - Holes might be tiny

RMS is neither optimal nor guaranteed to work even if CPU util is < 1

Given n invocations of `fork`, there will be $2^n - 1$ child processes

An I/O request would induce a process changing from running -> waiting.

For a logical address space of 16 bytes, among which there are 8 pages, and assuming the use of a page table:

- For the corresponding physical memory, each frame is 2 bytes
- The last byte of frame 1 has a corresponding logical memory address whose offset is 1
- The 3 highest bit of the logical address are used to index into the page table.

Reasoning:

$$\begin{aligned} 2^4 &\Rightarrow m = 4 \\ 2^3 &\Rightarrow n = 3 \text{ bits for page table index} \\ 16 \text{ bytes}/8 \text{ pages} &= 2 \text{ bytes per page } \textit{and} \text{ frame} \\ m - n &= 4 - 3 = 1 \text{ bit for the offset} \\ \text{Last byte of first frame} &= \underbrace{000}_{\text{page index}}1 \end{aligned}$$