Processes
Ch. 2

Process

A Program In Execution

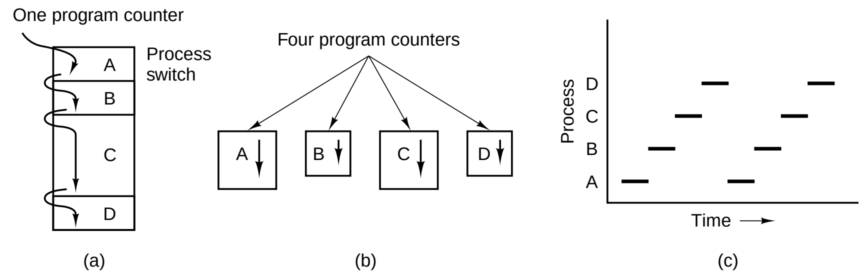
Program is a set of instructions.

Process is a programming being run:

- The program.
- The program's data.
  Memory, CPU registers, stack.
- The current location (PC).

Managing Processes

OS alternately works on multiple processes.



multiprogramming

Process Switching

OS is invoked by interrupt or syscall.

Save CPU registers, including PC and stack pointer.
Some will have been saved on the stack when the OS started.
Some still in the hardware registers.
Copy 'em all to some appropriate table in the OS.

Update VM tables.

Update OS records.

Copy the saved registers from memory into the hardware.
Restore the PC last, which transfers to the new pgm.

Process Creation

- System initialization.
- Spawned by request from an existing process.
- Created by user command (keyboard or mouse).
- Batch job submission.

Process creation is performed by the kernel.

Process creation is initiated by the kernel or another process.

Termination

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

Family Tree

When a process creates another, they are parent and child processes.

Treatment and privileges may depend on this family relationship.

The processes form a graph or tree.

Tree terminology applies: Processes may be ancestors, descendants, or siblings.

Unix uses this, and the root is process 1, init.

Windows does not impose this hierarchy, though a programmer is free to organize his processes this way.

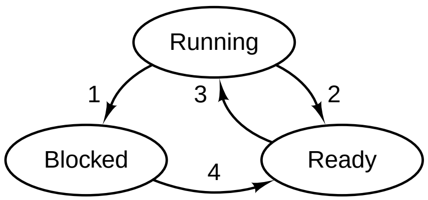
Process States

Each process has a state.

The OS changes the state of a process in response to an event.

The state indicates what the processes is allowed to do next.

The exact set of states used depends on the OS designer.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Implementation

OS Keeps a table of processes.

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Interrupts

Interrupts transfer control to the O/S, where:

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Threads: Split The Process

Processes Have:

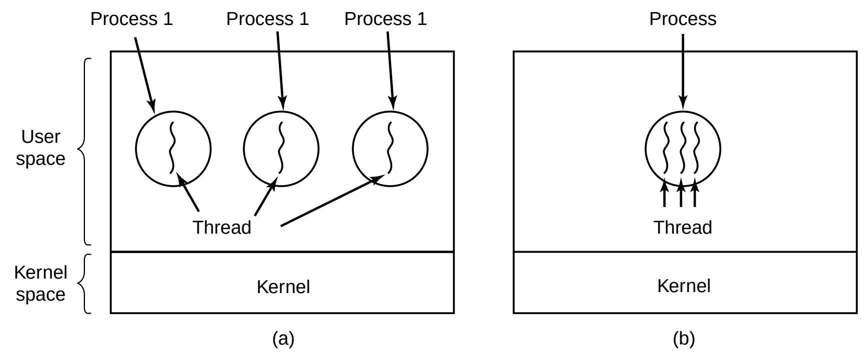
- Resources, especially memory
- A thread of execution

Divide these:

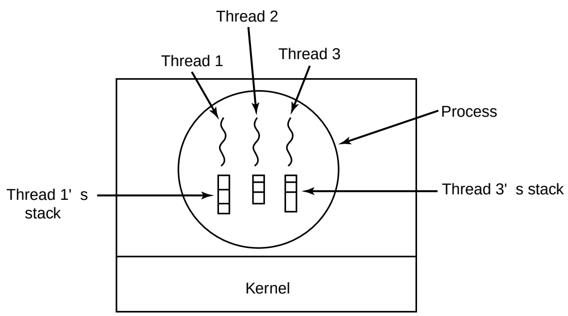
- The process (or *task*) owns the resources.
- Threads execute within tasks and use the task's resources.

Some authors favor the term task for the object that holds the resources, then one or more threads run in the task.

Threads and Processes



Each Thread Has Its Own Stack



Threads must each have their own copies of local data and call return locations.

They have different locations within the same memory image, and each thread may refer to stack data in other threads.

Threads v. Processes

Threads can communicate more efficiently.

Threads can clobber each other's variables.

Threads are cheaper to
create *destroy* *switch*

Thread States

Threads have states as processes do.

Running

Ready

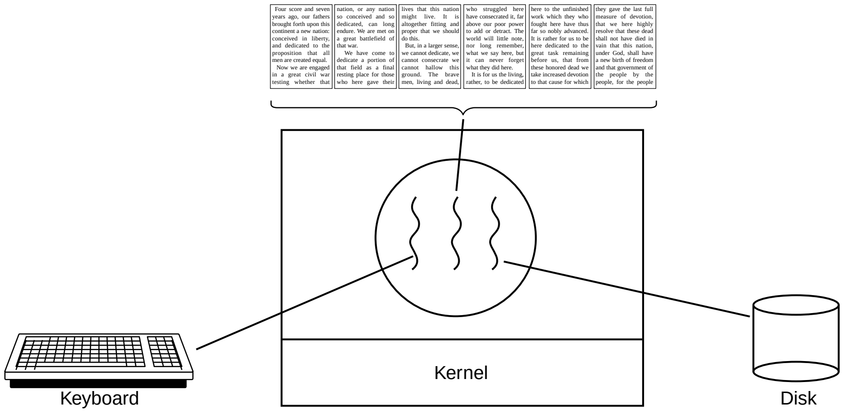
Blocked

Terminated.

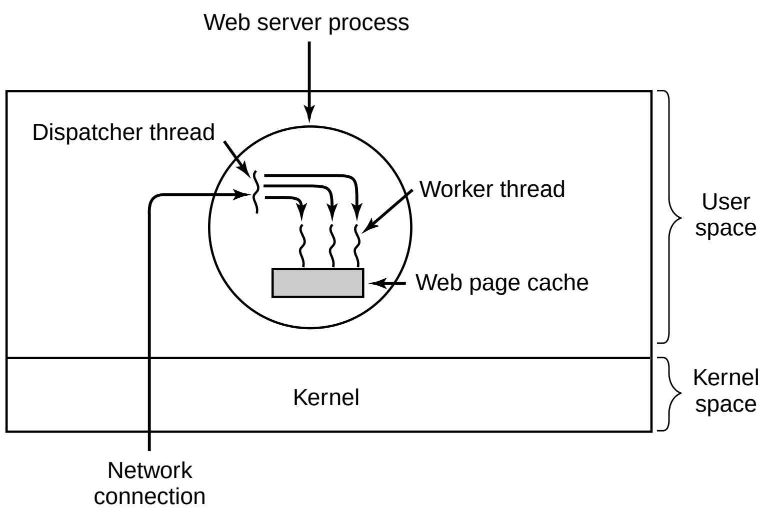
Thread Record-keeping

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Threaded Word Processor



Threaded Web Server



User Threads

Threads implemented by a library.
The kernel does not know about them.

The kernel knows about processes.

The library divides each process into one or more threads.

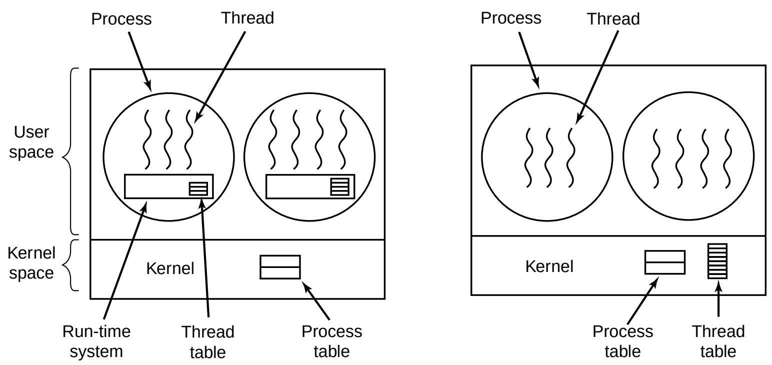
Threads scheduled by the library.
Library apportions time given by the kernel to the process.

Kernel Threads

Threads implemented by the kernel.
Each has an entry in a kernel table.

Threads scheduled by the kernel.

User and Kernel Threads



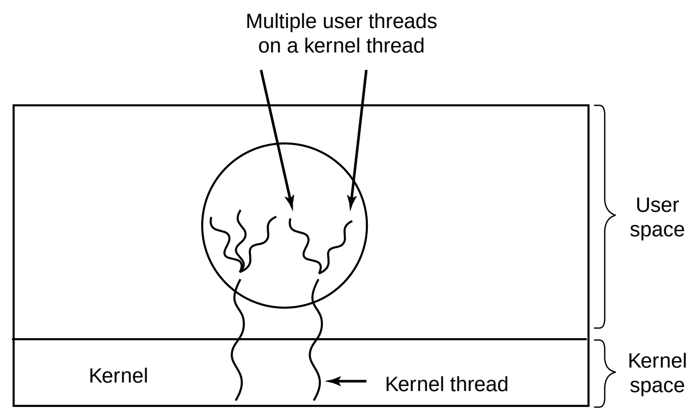
User v. Kernel Threads

User Threads	Kernel Threads
Write a library.	Modify the OS kernel.
The library can be portable.	Modify kernel on each OS.
Switching is fast.	Switching requires kernel intervention
Scheduling can be more easily tailored to the application	Scheduling is wired into the kernel

User v. Kernel Threads, Cont.

User Threads	Kernel Threads
OS Scheduling ignores threads	OS Scheduling accounts for threads
One thread blocks, all block	One thread blocks, others continue
All threads run on the same CPU	Can use SMP

Combined User and Kernel Threads



Combined Approach

Threads are created and scheduled by the kernel.

A library divides kernel threads into user threads.
These are created and scheduled by library code run inside a kernel thread.

Advantages of both.

Can be created atop any threaded kernel.

Solaris
Windows “fibers”

Other Hybrids

Scheduler activations: Kernel notifies the library when a thread blocks.

Pop-up: Thread created by an event such as a packet arrival.

Re-Writing for Threading

Unexpectedly difficult.

Global variables create race conditions.

Libraries often use globals.
Errno
Random number seed.

Library calls are often non-reentrant.
Malloc’s data structure.

Solutions

Forbid global variables.
Breaks much existing code.

Give separate copies of the “globals” to each thread.
No language support.

Allocate a block for this purpose pass to each call.
Yecch; works.

Interprocess Communication

Threads communicate through shared memory.

Depending on OS, processes may share memory areas.
A few selected; not the whole thing.

Through file system.

Message passing.
Includes Unix pipes.

Shared data structures subject to *race conditions.*

Process Progress

Processes proceed concurrently.

Uniprocessor: Interleaving.
Multiprocessor: Interleaving and parallelism.

Processes progress at different rates.

The execution sequence of statements in different processes is unpredictable.

In fact, statement A may be executed in the middle of the execution of statement B.

Shared data may be updated in unpredictable ways.

Update Example

Consider the following function shared by multiple processes.

```
void inc(int *ptr)
{
    int qty = *ptr;
    ++qty;
    *ptr = qty;
}
```

Machine language is close to this, regardless of source code.

Execution Order

Process P1	**Process P2**
•	•
qty = *ptr;	•
•	qty = *ptr;
•	++qty;
++qty;	•
*ptr = qty;	•
•	*ptr = qty;
•	•

Increment Failure

If `ptr` values are the same, one count is lost.

Eliminating `qty` does not help.
Think machine code level.

Creates an intermittent bug.
Probably the worst kind.

This is a race condition.

Locus of the problem: Shared data.

Solutions

If sharing is not needed, just make separate copies.
If the pointers differ, the problem disappears.

If sharing is required, synchronize the processes so that operations occur in a predictable order.

Sharing is needed when processes must cooperate.

Mutual Exclusion

Many shared data problems can be solved by mutual exclusion.

Mutual Exclusion.
The policy that when one process is using a particular resource, all others are excluded.

Critical Resource.
A resource which may be used by only one process at at time.

Critical Section (CS).
A portion of code that may be executed by only one process at a time.

Mutual Exclusion Methods

Disable interrupts.

Software lock variables.

Hardware-supported lock variables.

OS-supported operations.

Disable interrupts

Works on a uniprocessor:

```
/* Disable interrupts */
/* Critical section */
/* Enable interrupts */
```

May cause lost interrupts.

May reduce the responsiveness of an OS.

Not practical for user code.
Sometimes used for short waits in kernel.

Trend is away from OS's assuming a uniprocessor environment.

Lock Variable

```
int locked = 0;
```

Process 0

```
while(locked) /*wait*/;
locked = 1;
/* Crit. Sec. */
locked = 0;
```

Process 1

```
while(locked) /*wait*/;
locked = 1;
/* Crit. Sec. */
locked = 0;
```

Simple lock variable. Doesn't actually work.

Take Turns

```
int turn = 0;
```

Process 0

```
while(turn != 0)
   /* wait */;
/* Crit. Sec. */
turn = 1;
```

Process 1

```
while(turn != 1)
   /* wait */;
/* Crit. Sec. */
turn = 0;
```

Strict alternation: Process 0 may wait on 1, even if 1 does not need the CS at all.

Peterson's

For either of two processes:

```
shared int turn = 0;
shared int interested[2] = { 0, 0 };

int me = my_id(); /* 0 or 1 */
int other = 1 - me;
interested[me] = 1;
turn = me;
while(turn == me && interested[other]) /* loop */;
/* Critical */
interested[me] = 0;
```

Can be generalized for more than two processes.

Peterson's

The key is the assignment to `turn`.

If both perform it about the same time, exactly one will win.

Test and Set Lock

TSL reg, address

An *atomic* version of:

```
bool testset(int *target)
{
    bool ret = *target;
    *target = 1;
    return ret;
}
```

Implemented as a single, special hardware instruction, not an ordinary function.

TSL Fixes Lock Variable

```
int locked = 0;
```

Process 0	**Process 1**
`while(testset(&locked))`	`while(testset(&locked))`
`/*wait*/;`	`/*wait*/;`
`/* Crit. Sec. */`	`/* Crit. Sec. */`
`locked = 0;`	`locked = 0;`

Semaphore

An integer variable with these three extra operations:

- Initialize to a non-negative value.
- *down*: Decrement; if negative the caller is blocked.
- *up*: Increment; if result is non-positive, a waiting process is unblocked.

Most OS's and most thread libraries have support for semaphores.

Mutual Exclusion With Semaphores

```
semaphore s = 1;
...
down(s);
/* Critical Section */
up(s);
```

The initial value is the number of processes permitted in the CS simultaneously.

Limited Forms

A semaphore initialized to one and used for mutual exclusion is a *binary semaphore.*

A *mutex* is locked or unlocked; no counting.

Producer/Consumer Problem

One process is sending messages which another is reading.
Mostly any web application.

Messages reside in a queue between transmission and delivery.

The queue is of limited size.

The receiver must wait if the queue is empty.

The sender must wait if the queue is full.

Producer and Consumer Using Semaphores

```
semaphore mutex = 1;
semaphore avail_msgs = 0;
semaphore avail_space = BUF_SIZE;
queue<BUF_SIZE> buffer;
```

Producer

```
item = produce_item();
down(avail_space);
down(mutex);
queue.insert(item);
up(mutex);
up(avail_msgs);
```

Consumer

```
down(avail_msgs);
down(mutex);
item = queue.remove();
up(mutex);
up(avail_space);
consume_item(msg);
```

Monitors

Hoare

Collection of data and operations: A class.

One process at a time can be actively running any of its methods.

Condition variables:
$\mathit{wait}(c)$: Suspend caller on condition c.
$\mathit{signal}(c)$: Resume a suspended process (if any)

Monitors

The *signal* call is generally the last thing. If not, the caller is suspended in favor of the signaled process.

Brinch-Hansen proposes simply limiting *signal* to be the last operation.

Hoare didn't think of that, but all his examples worked that way.

Producer and Consumer Using Monitors

```
monitor ProducerConsumer {
   condition full, empty;
   int count = 0;
   msg queue<N> b;

 void insert(msg item)
 {
    if(count == N)
       wait(full);
    b.insert(item);
    count++;
    signal(empty);
 }

 msg remove() {
    if(count == 0)
       wait(empty);
    item = b.remove();
    count--;
    signal(full);
    return item;
 }

}
```


Last Slide v. Text Solution

When nothing is waiting on `c`
`signal(c)` is a no-op

The textbook conditions eliminate some `signals` which are no-ops.

The text solution adds a certain tidiness.

Both solutions may run no-op `signals`.

Monitors v. Semaphores

Monitors higher-level than semaphores.
Usually easier to code.

Semaphores easier to add to an existing language.

Neither is much use across a network.

Message Passing

send(*destination*, *message*)

receive(*source*, *message*)

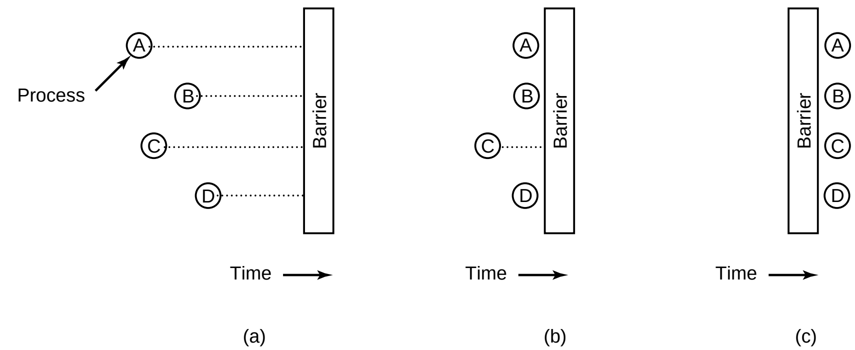
Makes more sense with networks.
Use acks to deal with message loss.
Authentication

There may or may not be buffer space to allow the sender to continue.

A receiver may wait until there is a message, or return with an error message.

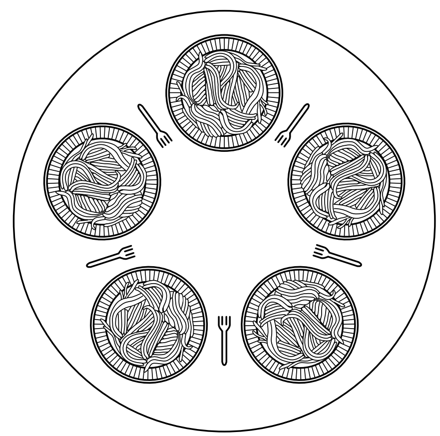
Barriers

Each process that reaches the barrier waits until all have.



Dining Philosophers

A classic problem in synchronization.
Due to Dijkstra



Dining Philosophers

There are five philosophers.

Their life consists of eating and thinking.

They can only afford five forks.

They each require two forks to eat. When one wants to eat, he must pick up his left and right forks. If either is in use, he must wait.

Problem: Synchronize the philosophers so they neither deadlock nor starve.

Non Solution

For philosopher i:

```
semaphore fork[5] = { 1, 1, 1, 1, 1 };
. . .
while(1) {
    /* Think */
    down(fork[i]);
    down(fork[(i+1) % 5]);
    /* Eat */
    up(fork[i]);
    up(fork[(i+1) % 5]);
}
```

Can deadlock.

Poor Solution

For philosopher i:

```
semaphore mutex = 1;
. . .
while(1) {
    /* Think */
    down(mutex);
    /* Eat */
    up(mutex);
}
```

Waste of precious forks.

Good Solution I

```
semaphore mutex = 1;
semaphore s[5] = { 0, 0, 0, 0, 0 };
enum { thinking, hungry, eating } state[5];
. . .
void test(int i) {
    if(state[i] == hungry &&
            state[(i-1)%5] != eating &&
            state[(i+1)%5] != eating) {
        state[i] = eating;
        up(s[i]);
    }
}
```

Good Solution II

```
while(1) {
    /* Think */
    down(mutex);
    state[i] = hungry;
    test(i);
    up(mutex);
    down(s[i]);
    /* Eat */
    down(mutex);
    state[i] = thinking;
    test((i-1)%N);
    test((i+1)%N);
    up(mutex);
}
```


Elegant Solution

For philosopher i:

```
semaphore fork[5] = { 1, 1, 1, 1, 1 };
semaphore room = 4;
. . .
while(1) {
    /* Think */
    down(room);
    down(fork[i]);
    down(fork[(i+1) mod 5]);
    /* Eat */
    up(fork[(i+1) mod 5]);
    up(fork[i]);
    up(room);
}
```


The Readers and Writers Problem

A collection of data is restricted by the following rules:

- Any number of process may simultaneously read the data.
- Only one process at a time may write the data.
- No process is allowed to read the data while it is being written by another process.

Problem occurs in databases.

Readers and Writers Using Semaphores

```
int readcount=0; semaphore mutex=1, writesem=1;
```

Reader

```
down(mutex);
readcount++;
if(readcount == 1)
   down(writesem);
up(mutex);
/* Read */
down(mutex);
readcount--;
if(readcount == 0)
   up(writesem);
up(mutex);
```

Writer

```
down(writesem);
/* Write */
up(writesem);
```

Writers can starve.

CPU Scheduling

When the CPU becomes idle, the OS must choose a job to run there.

When a new process is created, continue the parent or the child?

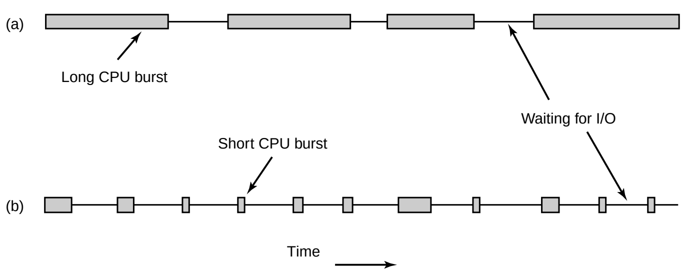
When the running process exits or blocks for I/O.

When a device interrupts the CPU.
Depends on type of scheduling algorithm.

Process Behavior

Jobs alternate between running and I/O waiting.

Programs are I/O-bound or compute-bound.



Treat each period of CPU activity as a unit: *burst*.

Properties

As CPUs get faster, jobs become more I/O-bound.

Generally, I/O-bound jobs are given priority, since they will run and leave soon.

Want to keep both the CPU and the I/O devices busy.

Categories

Batch

Interactive

Real time

Goals

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the 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
- Predictability - avoid quality degradation in multimedia systems

Preemption

Non-preemptive: A non-preemptive scheduling algorithm chooses which job gets the CPU, then lets it run to completion.

Preemptive: A preemptive scheduling algorithm can remove a job from the CPU before it is finished. It will be returned to the CPU later.

Batch Scheduling

Admission Scheduler: Admit a job to the system.

Memory Scheduler: Bring a process into, or remove a process from, main memory.
Control the *degree of multiprogramming*.

CPU Scheduler: Decide which of the processes in memory the CPU should execute. This is what we usually talk about.

Batch Scheduling Algorithms

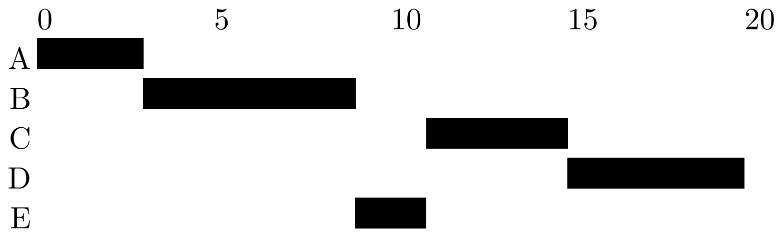
First-Come First-Served
I/O-bound jobs tend to be stuck in line, so devices are underused.

Shortest Job First
Reduces that problem
Non-preemptive.

Shortest Remaining Time Next
Preemptive
A new short job will be run immediately.

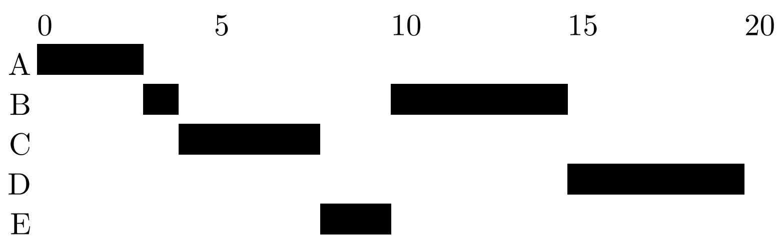
Shortest-Job First

Process	Arrival	Service
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Shortest Remaining Time Next

Process	Arrival	Service
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Interactive Scheduling Algorithms

Round-Robin

New jobs enter a queue, like FCFS.

Each job is given a *quantum*.
Runs until done, blocked, or the quantum expires.

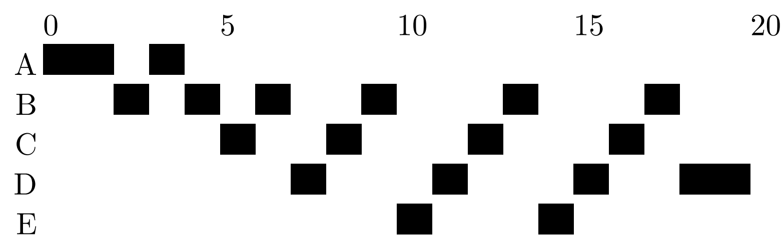
If quantum expires, returned to the end of the queue.

Quantum too short: Waste time switching.

Quantum too long: turns into FCFS.

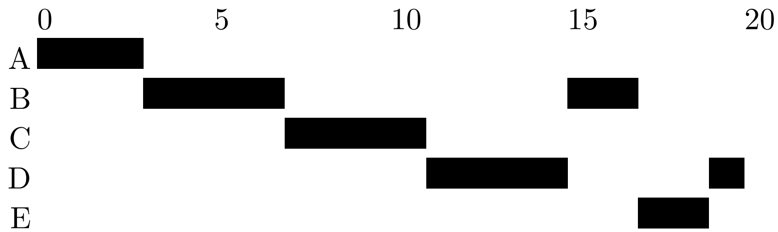
Round Robin, Quantum 1

Process	Arrival	Service
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Round Robin, Quantum 4

Process	Arrival	Service
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Priorities

Jobs have priorities:
Take the oldest job of highest priority next.

Static priorities: Administrative.

Dynamic priorities: Based on job's previous behavior.

Dynamic priority is typically lowered when a quantum expires.
Keeps response good for quick interactions.

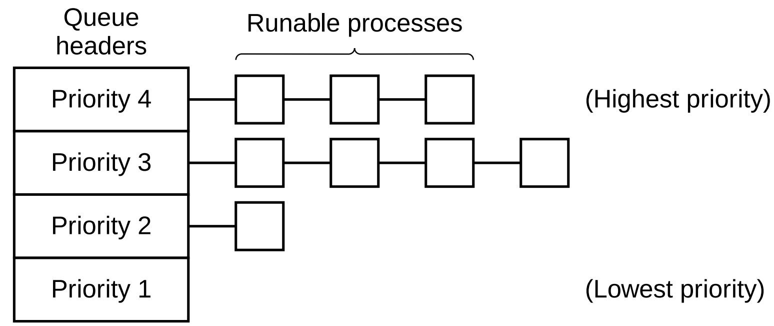
Longer quantum may be given at lower priority.
Lets CPU-bound jobs finish with fewer swaps.

Combination of static and dynamic often used.

Multiple Queues

A separate queue may be used for each priority.

Scheduler takes a job from the head of the highest priority non-empty queue.



Shortest Process Next

Using SJF interactively: must predict burst length.
Use lengths of the previous bursts from the same job.

T_i = Actual execution time of the ith burst.
S_0 = Predicted execution time of the first burst.
S_i = Predicted execution times of successive bursts.

Compute S_n from previous actual execution times:

$$S_0 = \text{arbitrary value}$$
$$S_{n+1} = aT_n + (1-a)S_n, n >= 1$$

The value of a, $0 \leq a \leq 1$, is an arbitrary parameter.

Larger a favors recent; smaller retains older values.

Run Time Prediction

$$S_{n+1} = aT_n + (1-a)aT_{n-1} + \dots$$
$$+ (1-a)^i aT_{n-i} + \dots + (1-a)^n S_1$$

$a = 1$: $S_{n+1} = T_n$

$a = 0.8$: $S_{n+1} =$
$0.8T_n + 0.16T_{n-1} + 0.032T_{n-2} + 0.0064T_{n-3} + \dots$

$a = 0.5$: $S_{n+1} =$
$0.5T_n + 0.25T_{n-1} + 0.125T_{n-2} + 0.03125T_{n-3} + \dots$

$a = 0$: $S_{n+1} = S_1$

Guaranteed Scheduling

Each of n processes get $1/n$ of the CPU.

A process which is t old should have t/n CPU time.

Keep track of actual CPU for each process, c.
Run the job with the lowest ratio c/n.

By always raising the lowest, they tend to stay together.

Lottery Scheduling

Each job gets some number of lottery tickets.

Scheduler chooses a ticket at random and runs the holder.

Jobs may exchange tickets.

Proportion of time given to each job can be well-controlled by the number of tickets.

Responsive: New jobs have the same chance as old ones with the same number of tickets.

Real-Time Scheduling

Events must be completed by some deadline.

Hard v. Soft

Hard: Missed deadlines are intolerable.
Think airplane controller.

Soft: Missed deadlines are unfortunate, but not fatal.
Think video stream.

Periodic v. Aperiodic

General purpose OSes have aperiodic load:
Work shows up when it shows up.

Many real-time apps are periodic:
Each task appears at a regular known interval.
Read the pressure every 4ms.

Schedulable Periodic Jobs

If there are m jobs, where the ith arrives each P_i time and takes C_i to run, then

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Or the schedule cannot be met.

For instance: reading the pressure each 4ms takes 1ms; reading the temperature each 6ms takes 2ms, and conditionally opening or closing the valve each 10ms takes 4ms. Then:

$$\frac{1}{4} + \frac{2}{6} + \frac{4}{10} = 0.983 \leq 1$$

Policy v. Mechanism

Trend is to separate these;
let the program control policy.

The OS handles scheduling.
The process can set the relative priorities of itself and its children.

Thread Scheduling

User threads: Kernel schedules the process, the library chooses which thread gets the service.

Library can use any of these methods, and may differ from the kernel.

Kernel threads: Kernel chooses the next thread.

May ignore which process the thread belongs to.

May not: switching within a process generally saves memory management overhead.

Sources

Tanenbaum, *Modern Operating Systems*
(Course textbook.)

The final Dining Philosopher's solution is presented in Stallings' *Operating Systems*, and mentioned by Silberschatz, Galvin and Gagne in *Operating System Concepts*.